A Denotational Approach to Measuring Complexity in Functional Programs

Kathryn Van Stone

June 2003

CMU-CS-03-150

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Stephen Brookes, Chair
Frank Pfenning
Dana Scott
Carl Gunter, University of Pennsylvania

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This research was funded in part through grants from the National Science Foundation.

Any opinions, findings and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect those of the sponsors.

Keywords: functional languages, denotational semantics, complexity, intensional semantics, call-by-value, call-by-name, operational semantics, time analysis

Abstract

Functional languages are considered useful in part because their applicative structure makes it easier to reason about the value returned by a program (its extensional behavior). When one wants to analyze intensional behavior, such as the time a program takes to run, some of the advantages of functional languages turn into disadvantages. In particular, most functional languages treat functional types as standard data types; however, the time it takes to use a functional type (that is, applying it to a value) depends not only on the type but also on the applied value. Also, a number of functional languages use lazy data types, so that we may need to handle cases where parts of a data value do not need to be evaluated. The time it takes to work with such data types depends not only on the type itself but how much of it has already been examined. It is particularly difficult to reason compositionally, in a syntax-directed manner, about intensional behavior of such programs.

In this dissertation we develop a compositional semantics for functional programs that allows us to analyze the time a program takes to run (in terms of the number of times certain operations are required). Our system uniformly handles both higher-order types and lazy types, by using internal costs (time needed to use an expression) as well as external costs (time needed to evaluation an expression). We create semantics for programs using either call-by-value evaluation strategy (where arguments are always evaluated) or call-by-name (where arguments are only evaluated when used). The technique we use to create the semantic functions is almost identical for both evaluation strategies and should be easily adjustable to other languages.

We then use the created semantics in several ways. First, we use the semantics to directly relate programs based on their intensional behavior. We also examine some example programs and compare their behavior with call-by-value and call-by-name evaluation. Lastly, we examine a more complex program, pattern matching using continuations, and derive some nonintuitive results.

Acknowledgements

I would like to thank some of those who made this work happen:

- My advisor, Stephen Brookes, for pointing me in this direction originally and for all the technical help needed to finish this project.
- My thesis committee, Frank Pfenning, Dana Scott, and Carl Gunter, for their patience and their support during my final few weeks.
- Sharon Burks, for handing all the administrative details involved in getting me my degree
- My parents, William and Joan Van Stone, for their help in keeping me moving and in proofreading the document.
- My sister, Lisa Van Stone, and my friend, Julia Smith, for their support in surviving the graduate school process.
- Finally, my husband Robert Smith, for his love, support, and endless patience during my long tenure as a graduate student.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Early low-level programming languages closely imitated the actions of the computer to simplify the translation of a program into machine instructions. Over time, however, our ability to translate programs has become more sophisticated, resulting in a number of programming languages designed to more closely match a specification of the desired result rather than a set of instructions. In particular, *functional languages* are designed so that programs resemble mathematical specifications more than machine instructions. Because of this property, it is often clear that a program will always return a correct result, but it may be more difficult to understand those internal properties that closely relate to machine instructions, such as the amount of time a program takes to run or the amount of memory needed in the process.

Functional languages typically share a number of properties. Perhaps the most important property of a functional language is the *applicative* structure of programs. Programs in a "pure" functional language typically consist of a series of function definitions. There are no assignment statements; instead of `for` or `while` loops a programmer uses recursion. Even in "impure" functional languages that also include assignment statements, programs are frequently written in such a style that assignments are the exception rather than the rule.

Another important aspect of functional programs is their treatment of functions as "first-class citizens." Many imperative languages treat procedures and functions differently from other data such as integers; for example, one cannot necessarily store a procedure in an array. In a functional language, however, functions are treated as data just as integers or lists are and can be passed to other functions as parameters, returned as results, or used as part of a data structure such as a list or tree. Apart from the simplicity (in concept) of treating all data types equally, such languages have several advantages. One is the ability to define general functions that manipulate data structures and then use them to create a wide variety of programs. For example, `map` applies a function to each element of a list; thus whenever we want a program that performs some conversion on each element of a list, we can simply use `map`. The function `map` is also *polymorphic* in that its behavior can be used for lists of any type (relative to the mapped function) and its behavior is essentially the same in each case. Another advantage is to be able to partially evaluate a procedure by applying it to some but not all of its arguments. The result is a procedure or function that takes the rest of the arguments. This can result in savings when a procedure is called many times with many of the same arguments.

There are some functional languages where functions are not first-class citizens; in particular, they do not allow functions to take other functions as arguments. Such languages are considered "functional" primarily because they do not allow assignments. These languages are called *first-order* functional languages; functional languages that allow functions as arguments are then called *higher-order* functional languages.

A third appealing property of functional languages (although not unique to them) is the widespread use of type constructors. Type constructors allow us to define data types without necessarily knowing much about the actual implementation; for example, we know that a list has a head, containing an element of some data type, and a tail, which is another list, and we can abstract away from implementation details. This facilitates analysis in two ways: by suppressing implementation details that do not affect the value of data, the analysis becomes much simpler; and by restricting the operations on data types we can guarantee that no element of the data type has an invalid form (such as a circular list). The drawback is that because implementation details are hidden, it may be harder to determine the amount of space they require or the time taken by operations on data.

Lastly, many functional languages allow *lazy* evaluation, either directly, as an integral part of the language, or indirectly, via higher-order types. When we speak of lazy evaluation, we usually mean at least one of two different optimizations. First, any evaluation is postponed until needed, avoiding possibly costly work if the result is never used. We also can manipulate data structures that are theoretically infinite as long as we only need to examine a finite amount of the structure. Second, evaluation can be *memoized*; that is, the result of an evaluation can be saved so that it can be looked up if needed again, avoiding re-evaluation.

Programmers and researchers thus often find functional languages very attractive. The applicative structure, lack of side effects, and use of flexible type constructors make it easier to design programs at a high level of abstraction, thus making it easier to prove program correctness. Furthermore, if we know that evaluation of one part of an expression has no side effects, it is easy to analyze the expression compositionally. It is also possible to substitute part of an expression with another of equal value and know that the overall result is unchanged; this property is known as *referential transparency*. Such properties make the *extensional* analysis (i.e., analysis of input-output properties) of functional programs reasonably simple. Additionally, the use of higher-order and lazy data types increases the expressive power and elegance of programs; a program written in a functional language is frequently shorter than the equivalent program written in an imperative language.

These advantages do come with a cost: As programs resemble function declarations more than machine instructions, it may be more difficult to determine the appropriate instructions needed for the computer to run them. Analyzing internal properties such as space and time complexity becomes more difficult when much of the work done by the computer is hidden from the programmer. Because the compiler or interpreter handles all space allocation invisibly, we may not know exactly how much space is allocated for a given data structure or how much time is taken managing the space. Furthermore, for certain types it may be difficult to define what we mean by time complexity. For example, consider the function

$$\texttt{twice} = \lambda f.\lambda x.f(f(x))$$

The time it takes to compute `twice` f x is dependent on the size of x, the complexity of f, and the size of the output of f on x. A standard definition for the complexity of an algorithm is a function from the size of the input to the time it takes to compute the output; however such a definition does not work for programs such as `twice`, which needs not only the complexity of f but the size of $f(x)$. A similar situation holds for lazy data structures; for example, we can easily show that a program that calculates the length of a list will take time proportional to the length of the list. Lazy lists, however, may not be completely evaluated, thus the time needed to find their lengths includes the time taken to finish evaluating the structure of the list, adding a potentially large amount of time. Furthermore, when various parts of an evaluation are delayed we may have difficulty predicting if and when evaluation will occur, adding to the problem of predicting the time it takes to evaluate a program.

1.1 Types of Semantic functions

1.1.1 Denotational vs. Operational

In this dissertation there will be two independent classifications of semantic functions. The first, common to most discussions of semantics is the distinction between *operational* semantics and *denotational* semantics. In general, an operational semantics for a programming language is provided as a collection of rules describing the steps taken by an abstract machine during expression evaluation. Many operational semantics come in one of two forms: *transition* (or single-step) semantics and *natural* (or big-step) semantics. A transition semantics for a language describes the rules for making a single action in the process of an evaluation. For example, if we let $e \longrightarrow e'$ mean that the expression e reduces to e' in one step, then the following could be rules for the evaluation of an application:

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2} \qquad \frac{}{(\texttt{lam}\ x.e)e_2 \longrightarrow [e_2/x]e}$$

where $[e_2/x]e$ indicates the expression e with e_2 substituted for all free occurrences of x. Conversely, a *natural* operational semantics describes how to find the result of an expression in one big step. The equivalent to the transition rules above would thus be

$$\frac{e_1 \Longrightarrow \texttt{lam}\ x.e \quad [e_2/x]e \Longrightarrow v}{e_1(e_2) \Longrightarrow v}$$

where $e \Longrightarrow v$ means that e evaluates to the (final) result v.

A denotational semantics, on the other hand, is a function mapping expressions to meanings, which typically are some form of mathematical entity. In particular, a denotational semantics forms the meaning of an expression from the *meanings* of its syntactic subparts. The natural deduction rule for application listed previously was not derived from the rules for evaluating e_1 and e_2, but from rules for e_1 and $[e_2/x]e$. If we say that $[\![e]\!]$ is the denotational meaning of an expression e, however, then the equivalent definition for an application would look something like the following:

$$[\![e_1(e_2)]\!] = [\![e_1]\!]([\![e_2]\!])$$

i.e., the meaning of $e_1(e_2)$ only requires the meanings of e_1 and e_2. This property, called *compositionality*, can often be an advantage to analysis as we do not have to account for anything but the expression itself and knowledge of the mathematical structure of the collection of meanings.

It is not always clear whether it is better to use an operational semantics or a denotational semantics. Depending on the exact language and semantics in use, either the denotational semantics or the operational semantics can be more intuitively "correct," thus in cases where the correctness of the definition is the goal, it is best to use the form that is more intuitively correct regardless of whether it is an operational or denotational semantics. In cases where the relation between the semantics and the computer instructions is important, an operational semantics is more likely to be the appropriate one to use. In cases where compositionality is a goal, a denotational semantics will be desired. If the desired semantics is not the one that is most obviously correct, another type of semantics will frequently be used as well, and then the two forms of semantics will need to be shown to correspond in some useful fashion. In this dissertation the operational semantics is arguably more intuitive and closer in spirit to machine-level implementation, but for ease of analysis we want a compositional semantics. Therefore we create a denotational semantics for analysis, and compare it with the operational semantics to show that our semantics is computationally reasonable.

1.1.2 Extensional vs. Intensional

Another way of classifying semantics concerns the level of abstraction at which a semantics is pitched. A semantics (operational or denotational) is *extensional* if it is primarily concerned with the final result and not other aspects of computation such as the time it takes to complete, the amount of space required, or the path traversed during computation. Conversely, a semantics is *intensional* precisely when it is concerned with such internal details. These two classifications are relative more than absolute, as the definition of "final result" and "internal details" may vary depending on the purpose of the semantic model.

Most traditional semantic models focus on extensional properties alone. Sometimes, to get a extensional definition, one must start with an intensional one and then abstract away the internal details, as was done for the example in the game semantics of [1]. In particular, a transition semantics technically is concerned with the path of a computation as well as the final result, but in practice when using a transition semantics we ignore the path and consider only the extensional properties. In this dissertation, however, the intensional properties are what is of interest so the semantics used must be intensional, although we also need to relate it to an extensional semantics.

1.2 Complexity versus Cost

Whenever we mention the *cost* of an expression or program, we are referring to a precise evaluation time, measured in some number of basic operations (recursive calls, CPU operations, etc.). Costs come in two varieties, the more abstract *macro cost* counts major operations such as function applications, and the more concrete *micro cost* counts low-level operations such as register accesses, which often better reflect the actual time it takes for the expression to be evaluated.

Complexity, on the other hand, refers to an asymptotic relation between the *measure* of the input to a function and the time it takes to determine the output. For example, the complexity of the reverse function for lists is typically written as $\mathcal{O}(n)$, where n is the length of the list, meaning that the cost of reversing the list is bounded by a linear function on n for sufficiently large lists. Those who analyze the efficiency of algorithms traditionally work with complexities because exact costs are frequently machine or compiler dependent. Language profilers, however, work with costs, primarily because determining costs is an easy task for a computer to do and in many cases the goal of a profiler is to find extra costs, such as constants costs, that asymptotic complexity analysis would not necessarily distinguish as significant.

In this dissertation we will concentrate on cost information rather than complexity. Our primary reason is one of feasibility: there is no standard for measuring the size of input for some data types, particularly higher-order types. We will, however, work only with macro costs, as our semantics would otherwise only be relevant to a particular machine whereas we want to find a more abstract notion of evaluation time. Furthermore, using macro costs, the semantic functions will be more easily converted to complexity functions (where it is possible to define complexity) because complexity functions also usually measure time by counting high-level operations. Our semantic functions will have sufficient mathematical structure so that measures and final costs can be derived from the more precise cost analysis. For example, the call-by-value semantics will tell us that the cost of the `length` function that computes the length of a list is

$$n(t_{+} + t_{\text{tail}} + t_{\text{false}}) + (n+1)(t_{\text{nil?}} + t_{\text{app}} \ + t_{\text{rec}}) + t_{\text{true}}$$

where n is the actual length of the list, and the costs represent the major actions of applying a function (t_{app}), unrolling a recursive definition (t_{rec}), taking the tail of a list (t_{tail}), taking either the false (t_{false}) or true (t_{true}) branches of a conditional, and adding two integers (t_{+}). From this information we can determine that the time it takes to run `length` is $\mathcal{O}(n)$, i.e., is proportional to the length of the list.

1.3 Evaluation Strategies

When evaluating an application, there is more than one way to handle the argument. The argument could be evaluated first and the result handed to the calling function. This type of evaluation is called *strict* or *call-by-value*. Alternatively, the argument could be saved and evaluated only when needed. If the argument is evaluated every time it is needed, the evaluation strategy is called *call-by-name*. If the result of evaluating the argument is saved so reevaluation is not necessary, the evaluation strategy is called *call-by-need*. Thus when evaluating the expression

$$(\texttt{lam}\ x.\texttt{lam}\ y.x + x)\ e_1\ e_2$$

The expression e_1 is evaluated once for call-by-value and call-by-need, but twice for call-by-name, while the expression e_2 is evaluated once by call-by-value, and not at all by call-by-need and call-by-name. If it takes a lot of time to evaluate e_1, then the call-by-value and the call-by-need strategies will take significantly less time to evaluate the overall expression. On the other hand, if it takes a long time to evaluation e_2, then the call-by-name and call-by-need evaluations strategies will be faster than the call-by-value.

It would thus seem that the call-by-need evaluation strategy would always be the most efficient. In practice, however, there is considerable overhead and complexity involved in keeping track of unevaluated expressions and updating the results of evaluation. For that reason there are many functional languages (e.g., Lisp, SML) that implement call-by-value evaluation. Even those languages that implement call-by-need (e.g., Haskell, Miranda) will often use strictness analysis to determine which arguments will always be evaluated, and use call-by-value evaluation on those arguments.

1.4 The Semantics

The goal of this work is the development of a denotational profiling semantics of higher-order strict and non-strict functional languages in order to analyze complexity behavior of programs in such languages. We are particularly interested in describing the costs of programs where part of the computation is delayed — either because an argument is needed or because a data type is lazy. The specific language in use throughout this work is relatively simple — essentially the λ-calculus with conditionals, recursion, and constants. This language is a generalization of PCF ([39]) in that we do not explicitly define the constants. It will be analyzed under two evaluation strategies: call-by-value and call-by-name. We discuss possibilities for analysis under call-by-need evaluation in section 6.2.4.

The desired semantics for the language under both evaluation strategies will satisfy the following properties:

Compositionality: The meaning of an expression will be dependent only on the meaning of its subparts. We can thus analyze the properties of an expression without needing to know in what context the expression occurs. In particular, the context can affect the cost associated with evaluating an expression under the call-by-name strategy, so the compositional semantics must contain sufficient information so that its meaning is still accurate regardless of the context.

Soundness: We will relate all the semantic functions to a relatively abstract operational model: one that other researchers ([37]) have studied. One particularly important relationship we will show is that the denotational semantics is sound. A denotational semantics is *sound* relative to an operational semantics if, whenever an expression e evaluates to v under the operational semantics, the (extensional) meanings of e and v are the same. With intensional semantics the relationship is more complicated; e.g., if e evaluates to v in time t in the intensional operational semantics, then the denotational meaning of e should be closely related to the meaning of v except that it should also include an additional cost of t. Once we define the intentional semantics we will make precise the meaning of "including an additional cost."

Adequacy: Another important relationship between the denotational and operational semantics concerns the behavior of programs that fail to terminate. A denotational semantics is *adequate* when a program fails to terminate in the operational model precisely when its denotational meaning indicates non-termination. With soundness and adequacy we can generally use the denotational model to analyze properties of programs and know that the properties hold for the operational model as well. In this dissertation the adequacy property will also ensure that, although the mathematical model may contain cost in unusual locations, the meaning of an expression will either indicate non-termination or be essentially a pairing of a final value and the cost it took to evaluate it.

Separability: Each intensional semantic function will have a related extensional semantics, one that gives the same final results but does not contain any consideration of costs. It is important to be able to recover this semantics to make sure that the intensional semantics treats the data the same way it would extensionally. We will be defining each semantic function such that we can easily obtain their extensional versions.

Furthermore, the semantic functions are *robust* and *flexible* in that the technique used to define the semantic functions applies to more than one operational semantics and more than one language. Thus the overall structure for both semantic functions will be very similar, each based on the same general cost construction. In addition we can easily add new constants, or adjust the meaning of old ones, and show that the new language is still sound, adequate, and separable. These properties indicate that other semantic functions may also be created to work with a variety of languages or operational definitions.

1.5 Related Work

When a program contains neither high-order nor lazy data types, we can analyze its complexity using many of the same techniques that have been used to analyze the complexity of imperative programs (e.g., see [2]). In these cases the applicative nature of functional programming is helpful, making it easy to convert a program to a functional description of its complexity. Some early work in automatic cost or complexity calculations ([50], [22]) took advantage of the structure of functional programs to develop recursive definitions of program complexity ([22]) or cost ([50]).

Other researchers ([6], [13], [37], [45]) have studied the costs of lazy first-order programs. Because not all elements of an expression are necessarily evaluated, lazy programs tend to be more difficult to analyze than their strict counterparts. The approach taken in these cases was to keep track of how much of the result of an expression is needed, based on the context in which the expression is used. The solutions were elegant for first-order functions but did not extend to higher-order types. The problem encountered was that the techniques for specifying that only a partial result was ever going to be used was inadequate when the data type was a function. Sands ([37]), did manage to get a higher-order approximation, but the approximation was sometimes very poor; Sands himself gives an example where the actual cost and the approximated cost can differ by almost an arbitrary amount. Sands did develop a semantics for analyzing macro costs of higher order lazy functional languages, but it was not compositional and did not handle memoization.

There have also been some examples of higher-order profiling semantics for call-by-value semantics ([41] and [12]). In [41], while Shultis did include some variant of internal costs for functional types (called *tolls*) the semantics was not proven sound relative to any model and was not sufficiently abstract to easily generalize to other systems. The first technique Gurr used in [12] is very similar to the technique used in this dissertation; we have managed, however, to generalize the technique to non-strict languages and make the technique more flexible. Gurr shifted instead to a more abstract system that was not only mathematically more complex and aimed at a different goal: comparing programs in different languages. Both Gurr and Shultis did not have any method to add costs except by primitive functions, and thus they were unable to track properties like the number of application or recursive calls, which, in practice, are often more relevant than the number of integer additions.

Greiner ([10]) created profiling semantics for a higher-order call-by-value language. His goal was to describe time and space costs for parallel languages. His techniques worked well for that goal but do not address the goals of this thesis; his semantics was not compositional nor did it handle any form of laziness outside of high-order types.

The primary way our semantic functions differ from their predecessors is in the use of an explicit *internal* cost. Unlike *external* cost, which refers to the cost of a computation, internal cost refers to the cost of *delayed* computations. This includes both the cost of applying a function (as opposed to the cost of evaluating the function itself) and the cost of evaluating the subparts of lazy data structures such as lazy lists. While other researchers ([41], [12]) included some form of internal cost, it only applied to function types. Furthermore, the meanings of expressions combined data and internal cost in such a way that it was not clear that the original extensional meaning was maintained. With separability, we can clearly determine which part of the meaning of a expression is related to internal cost and which to extensional data, and thus also ensure that the intensional semantics is consistent with an extensional semantics.

1.6 About this Document

In this dissertation we analyze two evaluation strategies. Chapters 3 and 4 cover the semantic definitions and the proofs that the properties described in section 1.4 hold for both strategies. In Chapter 2 we define the language itself and its extensional behavior, both operationally, with a traditional natural semantics, and denotationally. To make the transition to a profiling semantics easy and clear, we first define the denotational semantics using standard techniques. We then convert the semantics to an equivalent one that is significantly easier to extend to an intensional semantics. This chapter also includes the background for mathematical concepts used later.

In Chapter 3 we concentrate on the call-by-value evaluation strategy. First we provide an operational semantics including cost, similar to the one without cost but including a specific cost for each transition. Next we present the motivation and requirements for a particular cost structure used throughout the dissertation, followed by the profiling semantics. We show that this semantics is sound, adequate, and separable. We also show that there is a concrete example of a cost structure satisfying all the necessary requirements. At the end of the chapter we examine some simple programs to see how the semantics can be used to analyze complexity.

In Chapter 4 we consider the call-by-name evaluation strategy. The primary emphasis in this chapter is defining a sound, adequate, and extensional semantics for the call-by-name evaluation strategy using the same cost structure and general techniques as we did for the call-by-value evaluation strategy. The examples at the end of the chapter not only demonstrate how to use the semantics to analyze complexity but also demonstrate some problems that occur when evaluating the cost of partially evaluated data structures.

In Chapter 5 we use the profiling semantics to analyze the complexity of a non-trivial program, one that performs regular expression matching using high-order functions. We show that the complexity of this program is interesting and has some non-intuitive properties.

Chapter 6 summarizes the previous three sections and explores some possibilities for future work. We also discuss the analysis of semantics emulating the call-by-need evaluation strategy and show both the problems with our approaches and some suggestions for further research.

Chapter 2

The Extensional semantics

2.1 Introduction

One purpose of a profiling semantics is to provide a better understanding of the behavior of various programs (such as the `twice` program given in the introduction). Thus we will need to be able to correlate the intensional description of a program with its extensional meaning to ensure that our semantics is sensible. We must also ensure that the language in use has enough structure to define interesting programs. For that reason the language in this paper has abstractions (to form higher order functions), recursion (to allow programs with interesting profiling behavior), and some form of either pattern matching or conditionals (to allow recursive programs with base cases).

We thus work with a standard simply-typed functional language, similar to a subset of ML, with conditionals, recursion and primitive constants. It is essentially a generalization of PCF ([39]) where the constants are not initially specified; the definitions of the primitive constants are separate from the definitions of the rest of the language. This allows us to choose constants and data types as needed for examples without changing the overall proofs of soundness and adequacy; instead, we only need show that the new constants and data types satisfy certain assumptions.

In section 2.2 we give the formal definition of the language. We then, in section 2.3, give an operational semantics for both the call-by-value and the call-by-name evaluation strategies. In section 2.4 we give the definition of the denotational semantics. We begin with the standard fixed-point account of recursion using domains. This version, however, is too specific; we need a more abstract definition that will better enable us to add abstract cost. Therefore we adjust the semantics in two ways: using category theory for the semantic definitions, and a *monad* to handle the treatment of non-termination. Both adjustments lead to a more abstract definition that is based on a collection of properties sufficient to form sound and adequate semantic functions. We also show, in section 2.4.6, that the usual domain-theoretic semantics can be recovered by making the obvious choice of category and monad.

Soundness and adequacy are as important for the extensional semantics as for the intensional, so in sections 2.5 and 2.6 we discuss these properties. In chapters 3 and 4 we show that the soundness of the extensional semantics can be derived from the soundness of the intensional semantics and from separability. Thus in this chapter we only state the soundness properties and defer the proofs of soundness until later chapters. We do, however, prove adequacy in this chapter because the adequacy of the extensional semantics is needed to prove the adequacy of the intensional semantics. In later chapters we show how we use the adequacy of the extensional semantics and separability of the intensional semantics to prove adequacy of the various intensional semantics.

In the last section of this chapter we give the extensional categorical semantics for a number of constants, chosen either to aid in examples or to justify the assumptions made about them.

Most of the background theory covered in this chapter is described in either [11] (for the operational and domain-theoretic semantics) or [3] (for the category theory).

2.2 Language syntax

Because we treat constants separately from the rest of the language, the language syntax listed here is technically for a family of languages, not for a single language. Thus we will be able to use rather complicated languages for examples without needing the extra complexity for the more central proofs. These proofs will generally be based on certain assumptions made about constants; after the proofs we also show that many commonly used constants satisfy those assumptions.

Formally, we define the syntax of the language(s) as follows:

$$\begin{array}{rcl} e & ::= & x \mid c \mid e(e) \mid \texttt{lam}\ x.e \mid \texttt{rec}\ x.e \\ & & \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \end{array}$$

where c ranges over some set **Const** of constants. As the language contains conditionals we need at least two constants, `true` and `false`.

For each constant, let $\mathrm{ar}(c)$ be its *arity*, that is, the number of arguments needed; we assume that each constant has only one arity. Let $\mathrm{ar}(\texttt{true})$ and $\mathrm{ar}(\texttt{false})$ each be 0.

Next let **Construct** $\subseteq$ **Const** be a set of constants designated as constructors. These constants are used to form values; for example, `pair` is a constructor so `pair` $v_1 v_2$ is a value whenever v_1 and v_2 are values. In contrast, `fst` is not a constructor, so `fst` v will never be a value. As all constants of arity 0 form values, we consider them all to be constructors.

All constructors of arity 1 or more are either *strict* or *lazy*. Strict constructors evaluate each argument and form values from the results, while lazy constructors form values from their arguments without evaluating them. The distinction is only relevant for call-by-name evaluation; in call-by-value evaluation we treat all constructors as strict. Let **S-Construct** be the set of strict constructors and **L-Construct** be the set of lazy constructors.

It is not critically important that the language be strongly typed; in [6] the authors performed a similar study (first order only) on a Lisp-like language without types. Types, however, are still very useful. The primary reason we use types is to simplify the denotational semantics; we will never need to consider the behavior of incorrectly typed expressions such as `true` 3. Furthermore, the simply-typed λ-calculus with recursion is well studied, thus making it easy to match extensional behavior.

In order to give types to expressions, however, there needs to be a general method for including types, such as products and lists, without restricting the definition to those specific types. Therefore let δ range over a set of type constructors, i.e., partial functions from types to types. For example, the function $\delta_\times(\tau_1, \tau_2)$ is $\tau_1 \times \tau_2$, while the function $\delta_{\mathrm{list}}(\tau)$ converts a type τ to the type of lists where the elements of each list has type τ. We also use several ground types, represented as g; one such type that we must include is **bool** for handling conditionals.

Formally the set of types τ is defined as follows:

$$\tau \quad ::= \quad g \mid \tau \to \tau \mid \delta(\tau_1, \ldots, \tau_n)$$

To give types for the constants, for each type τ let $\mathbf{Const}_\tau$ be the set of constants having type τ. We require that $\mathbf{Const} = \bigcup_\tau \mathbf{Const}_\tau$; that is, each constant c has at least one type. We do not, however, require that each $\mathbf{Const}_\tau$ be disjoint from each other; a constant can be polymorphic and have more than one valid type. For example, for any pair of types τ_1, τ_2, `fst` can have type $(\tau_1 \times \tau_2) \to \tau_1$. $\mathbf{Const}_{\mathbf{bool}} = \{\mathtt{true}, \mathtt{false}\}$ and `true`, `false` are in no other set $\mathbf{Const}_\tau$.

The types for each constant must match their arity. Therefore if $c \in \mathbf{Const}_{\tau_1 \to \ldots \to \tau_n \to \tau}$, where τ is not a functional type, then c must have arity n.

A type of an expression is dependent on a type environment $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ which assigns types to variables. We assume that the x_i's are always distinct. If $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, and x does not equal any x_i in Γ, then $\Gamma, x : \tau = x_1 : \tau_1, \ldots, x_n : \tau_n, x : \tau$.

We say that an expression e has type τ under the environment Γ if the expression $\Gamma \vdash e : \tau$ is derivable from the rules listed in Figure 2.1. We say that e has type τ if $\vdash e : \tau$. We use the notation $x : \tau \in \Gamma$ to mean that $\Gamma = \Gamma', x : \tau, \Gamma''$ for some type assignments Γ' and Γ''. It is quite possible for an expression to have more than one type; the expression $\mathtt{lam}\ x.x$ has type $\tau \to \tau$ for any type τ.

A *program* is a typeable closed expression. When defining constants it is permissible to place additional restrictions on programs; for example, we can declare that a program may not contain certain constants even though such constants may be used for operational evaluation.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{c \in \mathbf{Const}_\tau}{\Gamma \vdash c : \tau}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \mathtt{lam}\ x.e : \tau' \to \tau} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1(e_2) : \tau}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{rec}\ x.e : \tau} \qquad \frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau}$$

Figure 2.1: Typing rules for functional languages

2.2.1 An example language

Simply assuming certain general properties of the constants will make the central proofs of soundness and adequacy more concise; however, in any specific instance, using a given collection of constants we will need to prove that the assumptions hold. To show that the assumptions are reasonable we need to indicate that the assumptions are likely to hold for constants we are likely to encounter. Therefore in this section we define a particular language, called FL (**F**unctional **L**anguage). FL is similar to the well-known language PCF ([39]), except that instead of the integer constants `pred` and `succ` it includes the standard arithmetic constants plus products, sums and lists. With these types we can demonstrate that the assumptions that we make about constants are likely to be reasonable because many commonly used data types are closely related to some combination of products, sums, and lists.

The ground types for FL are **bool** and **nat**. In addition there are three type constructors: the binary type constructors, $\times$ and $+$, plus the unary type constructor, **list**. For convenience, we will write the binary type constructors in infix notation; i.e. $\tau_1 \times \tau_2$ instead of $\times(\tau_1, \tau_2)$.

The constants are listed in Figure 2.1, where n can be any natural number, and τ, τ_1 and τ_2 can be any type. We write $\overline{n}$ for the syntactic numeral corresponding to n.

The constructors are `pair`, `inl`, `inr`, and `cons`. When examining the call-by-name or call-by-need evaluation strategies, `cons`, `inl`, and `inr` are lazy constructors and `pair` is a strict constructor.

Constant	Arity	Type
Integer Constants		
$\overline{n}$	0	$\mathbf{nat}$
$\leq$	2	$\mathbf{nat} \to \mathbf{nat} \to \mathbf{bool}$
$=$	2	$\mathbf{nat} \to \mathbf{nat} \to \mathbf{bool}$
$+$	2	$\mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$
$\dot{-}$	2	$\mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$
$\times$	2	$\mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$
Product Constants		
`pair`	2	$\tau_1 \to \tau_2 \to \tau_1 \times \tau_2$
`fst`	1	$\tau_1 \times \tau_2 \to \tau_1$
`snd`	1	$\tau_1 \times \tau_2 \to \tau_2$
Sum Constants		
`inl`	1	$\tau_1 \to \tau_1 + \tau_2$
`inr`	1	$\tau_2 \to \tau_1 + \tau_2$
`case`	3	$\tau_1 + \tau_2 \to (\tau_1 \to \tau) \to (\tau_2 \to \tau) \to \tau$
List Constants		
`nil`	0	$\mathbf{list}(\tau)$
`cons`	2	$\tau \to \mathbf{list}(\tau) \to \mathbf{list}(\tau)$
`head`	1	$\mathbf{list}(\tau) \to \tau$
`tail`	1	$\mathbf{list}(\tau) \to \mathbf{list}(\tau)$
`nil?`	1	$\mathbf{list}(\tau) \to \mathbf{bool}$

Table 2.1: FL Constants, their arities and types

We use the following syntactic sugar with FL:

- Integer addition, multiplication, etc., are written as $e_1 + e_2$, $e_1 \times e_2$, etc., instead of $+(e_1)(e_2)$.
- The case statement $\texttt{case}\ e_1 e_2 e_3$ is written as $\texttt{case}\ e_1\ \texttt{of}\ \texttt{left} : e_2\ \texttt{right} : e_3$.
- The product expression $\texttt{pair}\ e_1 e_2$ is written as $<e_1, e_2>$.
- List consing ($\texttt{cons}\ e_1 e_2$) is written as $e_1{::}e_2$ and we assume that :: associates to the right, i.e., $e_1 :: e_2 :: e_3 = e_1{::}(e_2{::}e_3)$. Also, the notation $[e_1, e_2, \ldots, e_n]$ is used to represent the list $e_1 :: e_2 :: \ldots :: e_n :: \texttt{nil}$.

2.3 The extensional operational semantics

The call-by-value operational semantics is listed in Figure 2.2 and the call-by-name operational semantics is listed in Figure 2.3. We write $e \Rightarrow_{\mathrm{v}} v$ to mean that e evaluates (operationally) to v using the call-by-value evaluation strategy; similarly, we write $e \Rightarrow_{\mathrm{n}} v$ to mean that e evaluates to v using the call-by-name evaluation strategy. Both operational semantics closely resemble standard natural semantics given for each evaluation strategy, as seen in [11]. We chose this particular form of abstract operational semantics to make it easy to add costs to cover higher level operations (such as the application of a function) rather than lower level operations (such as the access of a memory location).

The primary difference between these semantic definitions and those in sources like [11] concerns the treatment of constants. As we are separating constants in the definitions and proofs, there are separate judgments for each constant c of the form

$$\mathrm{vapply}(c, v_1, \ldots, v_i) \Rightarrow v \quad \text{and} \quad \mathrm{napply}(c, e_1, \ldots, e_i) \Rightarrow v$$

where i is some integer less than or equal to the arity of c. In each case the judgment means that c applied to the given arguments ($v_1, \ldots, v_i$ or $e_1, \ldots, e_i$) evaluates (using the respective evaluation strategy) to v. For the call-by-value semantics, when a constant c is applied to an expression e, e is first evaluated to a value v, then the judgment $\mathrm{vapply}(c, v) \Rightarrow v'$ is used to determine the effect of applying c to v. When there is more than one argument, each e_i is evaluated, but $cv_1 \ldots v_i$ evaluates to itself when the v_i's are results of evaluations and when i is less than the arity of c. Eventually we encounter a judgment of the form $\mathrm{vapply}(c, v_1, \ldots, v_n) \Rightarrow v$, which contains the final result. For call-by-name, the individual arguments are not evaluated when forming the intermediate results. Thus for call-by-name, $ce_1 \ldots e_i$ evaluates to itself when i is less than the arity of c. When all the arguments are available, we use a judgment of the form $\mathrm{napply}(c, e_1, \ldots, e_n) \Rightarrow v$; the proof tree containing this judgment may or may not contain evaluations of some or all of the e_i's.

As we have not specified all of the constants, we cannot list all the rules for constant application judgments; nevertheless, as seen in Figure 2.4 (for call-by-value) and Figure 2.5 (for call-by-name), some rules are standard. For example, no evaluation occurs for partially applied constants, so we can define a set of rules for them. Additionally, if c is a constructor, then its only action is to form a value.

We assume that all other rules have the form

$$\frac{e'_1 \Rightarrow_{\mathrm{v}} v'_1 \ldots e'_k \Rightarrow_{\mathrm{v}} v'_k}{\mathrm{vapply}(c, v_1, \ldots, v_n) \Rightarrow v}$$

$$\frac{}{c \Rightarrow_{\mathrm{v}} c} \qquad \frac{}{\mathtt{lam}\ x.e \Rightarrow_{\mathrm{v}} \mathtt{lam}\ x.e}$$

$$\frac{e_1 \Rightarrow_{\mathrm{v}} \mathtt{lam}\ x.e' \quad e_2 \Rightarrow_{\mathrm{v}} v' \quad [v'/x]e' \Rightarrow_{\mathrm{v}} v}{e_1(e_2) \Rightarrow_{\mathrm{v}} v} \qquad \frac{[\mathtt{rec}\ x.e/x]e \Rightarrow_{\mathrm{v}} v}{\mathtt{rec}\ x.e \Rightarrow_{\mathrm{v}} v}$$

$$\frac{e_1 \Rightarrow_{\mathrm{v}} \mathtt{true} \quad e_2 \Rightarrow_{\mathrm{v}} v}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Rightarrow_{\mathrm{v}} v} \qquad \frac{e_1 \Rightarrow_{\mathrm{v}} \mathtt{false} \quad e_3 \Rightarrow_{\mathrm{v}} v}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Rightarrow_{\mathrm{v}} v}$$

$$\frac{e_1 \Rightarrow_{\mathrm{v}} c v_1 \ldots v_i \quad e_2 \Rightarrow_{\mathrm{v}} v_{i+1} \quad \mathrm{vapply}(c, v_1 \ldots, v_{i+1}) \Rightarrow v}{e_1(e_2) \Rightarrow_{\mathrm{v}} v}$$

Figure 2.2: Operational semantics for a call-by-value functional language

$$\frac{}{c \Rightarrow_{\mathrm{n}} c} \qquad \frac{}{\mathtt{lam}\ x.e \Rightarrow_{\mathrm{n}} \mathtt{lam}\ x.e}$$

$$\frac{e_1 \Rightarrow_{\mathrm{n}} \mathtt{lam}\ x.e' \quad [e_2/x]e' \Rightarrow_{\mathrm{n}} v}{e_1(e_2) \Rightarrow_{\mathrm{n}} v} \qquad \frac{[\mathtt{rec}\ x.e/x]e \Rightarrow_{\mathrm{n}} v}{\mathtt{rec}\ x.e \Rightarrow_{\mathrm{n}} v}$$

$$\frac{e_1 \Rightarrow_{\mathrm{n}} \mathtt{true} \quad e_2 \Rightarrow_{\mathrm{n}} v}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Rightarrow_{\mathrm{n}} v} \qquad \frac{e_1 \Rightarrow_{\mathrm{n}} \mathtt{false} \quad e_3 \Rightarrow_{\mathrm{n}} v}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Rightarrow_{\mathrm{n}} v}$$

$$\frac{e_1 \Rightarrow_{\mathrm{n}} c e'_1 \ldots e'_i \quad \mathrm{napply}(c, e'_1, \ldots, e'_i, e_2) \Rightarrow v}{e_1(e_2) \Rightarrow_{\mathrm{n}} v}$$

Figure 2.3: Operational semantics for call-by-name functional language

$$\frac{i < \mathrm{ar}(c)}{\mathrm{vapply}(c, v_1, \ldots, v_i) \Rightarrow cv_1 \ldots v_i} \qquad \frac{n = \mathrm{ar}(c) \quad c \in \mathbf{Construct}}{\mathrm{vapply}(c, v_1, \ldots, v_n) \Rightarrow cv_1 \ldots v_n}$$

Figure 2.4: Standard rules for $\mathrm{vapply}(c, v_1, \ldots, v_n) \Rightarrow v$

$$\frac{}{\mathrm{napply}(ce_1, \ldots, e_i) \Rightarrow ce_1 \ldots e_i} \ (i < \mathrm{ar}(c))$$

$$\frac{}{\mathrm{napply}(ce_1, \ldots, e_{\mathrm{ar}(c)}) \Rightarrow ce_1 \ldots e_{\mathrm{ar}(c)}} \ (c \in \mathbf{L\text{-}Construct})$$

$$\frac{e_1 \Rightarrow_{\mathrm{n}} v_1 \ldots e_n \Rightarrow_{\mathrm{n}} v_n}{\mathrm{napply}(c, e_1, \ldots, e_n) \Rightarrow cv_1 \ldots v_n} \ (c \in \mathbf{S\text{-}Construct}, n = \mathrm{ar}(c))$$

Figure 2.5: Standard rules for $\mathrm{napply}(c, e_1, \ldots, e_n) \Rightarrow v$

for call-by-value, or

$$\frac{e'_1 \Rightarrow_{\mathrm{n}} v'_1 \ldots e'_k \Rightarrow_{\mathrm{n}} v'_k}{\mathrm{napply}(c, e_1, \ldots, e_n) \Rightarrow v}$$

for call-by-name, where n is the arity of c and the e'_j and v'_j can be any expression, although typically there will be some relation between the e'_j and the arguments to c. All this assumption means is that any premises to a rule (apart from side conditions) are straight evaluations and that there are no other rules for insufficiently applied constants. For example, one rule for application of integer addition in FL (using call-by-value) is

$$\mathrm{vapply}(+, \overline{2}, \overline{4}) \Rightarrow \overline{6}$$

More generally, for any natural numbers n, m,

$$\mathrm{vapply}(+, \overline{n}, \overline{m}) \Rightarrow \overline{n+m}$$

All the call-by-value addition rules are covered by the preceding rule because we use the constant application judgment only with fully evaluated arguments. For the call-by-name case, however, the arguments are unevaluated so the equivalent rule for addition is

$$\frac{e_1 \Rightarrow_{\mathrm{n}} \overline{n} \quad e_2 \Rightarrow_{\mathrm{n}} \overline{m}}{\mathrm{napply}(+, e_1, e_2) \Rightarrow \overline{n+m}}$$

2.3.1 Properties of the operational semantics

There are some critical properties of both operational semantics which we will need before we show that the denotational semantics is sound. The main two, type soundness and value soundness, are fundamental properties but do require assumptions about the unspecified rules concerning constant application.

There are three possible ways to define a *value* relative to an operational semantics. The first, and most fundamental, is that a value is the result of an evaluation. The second is that a value

$$\frac{}{\mathbf{v\text{-}value}\ c} \qquad \frac{}{\mathbf{v\text{-}value}\ \mathtt{lam}\ x.e}$$

$$\frac{\mathbf{v\text{-}value}\ cv_1 \ldots v_{i-1} \quad \mathbf{v\text{-}value}\ v_i}{\mathbf{v\text{-}value}\ cv_1 \ldots v_i}\ (i < \mathrm{ar}(c))$$

$$\frac{\mathbf{v\text{-}value}\ cv_1 \ldots v_{n-1} \quad \mathbf{v\text{-}value}\ v_n}{\mathbf{v\text{-}value}\ cv_1 \ldots v_n}\ (n = \mathrm{ar}(c) \text{ and } c \in \mathbf{Construct})$$

Figure 2.6: Values of the call-by-value semantics

$$\frac{}{\mathbf{n\text{-}value}\ c}$$

$$\frac{}{\mathbf{n\text{-}value}\ \mathtt{lam}\ x.e}$$

$$\frac{}{\mathbf{n\text{-}value}\ ce_1 \ldots e_i}\ (i < \mathrm{ar}(c))$$

$$\frac{}{\mathbf{n\text{-}value}\ ce_1 \ldots e_{\mathrm{ar}(c)}}\ (c \in \mathbf{L\text{-}Construct})$$

$$\frac{\mathbf{n\text{-}value}\ v_1 \ldots \mathbf{n\text{-}value}\ v_{\mathrm{ar}(c)}}{\mathbf{n\text{-}value}\ cv_1 \ldots v_{\mathrm{ar}(c)}}\ (c \in \mathbf{S\text{-}Construct})$$

Figure 2.7: Values of the call-by-name semantics

is an expression that evaluates to itself. The last is that a value is an expression that satisfies some explicitly defined rules. An operation semantics is *value sound* if the first two definitions are equivalent; i.e., if e evaluates to a value v, then v must evaluate to v as well. This property holds for both the call-by-value and call-by-name semantics if one assumes that any additional rules for constants are also value sound.

We can also explicitly give the definition of a value for both the call-by-value and call-by-name semantics. Because the operational semantics are different there are two definitions for values. For the call-by-value semantics we say that v is a value if $\mathbf{v\text{-}value}\ v$ holds, where Figure 2.6 contains the rules for $\mathbf{v\text{-}value}\ v$. Similarly, for the call-by-name semantics, v is a value if $\mathbf{n\text{-}value}\ v$ holds, where Figure 2.7 contains the rules for $\mathbf{n\text{-}value}\ v$.

For both evaluation strategies it is easy to show (by induction on the structure of evaluation) that if $e \Rightarrow_{\mathrm{v}} v$ then $\mathbf{v\text{-}value}\ v$ and if $e \Rightarrow_{\mathrm{n}} v$ then $\mathbf{n\text{-}value}\ v$. Similarly it is not difficult to show that if $\mathbf{v\text{-}value}\ v$ then $v \Rightarrow_{\mathrm{v}} v$ and if $\mathbf{n\text{-}value}\ v$ then $v \Rightarrow_{\mathrm{n}} v$. Thus all three definitions for value are equivalent.

It is also true that any value for the call-by-value semantics is also a value for the call-by-name semantics. This does not necessarily hold in the other direction.

The other critical property needed is type soundness. An operational semantics is type sound if for all closed expressions e whenever e evaluates to v and has type τ, then v must have type τ

as well. Formally, the call-by-value semantics is type sound if for all closed expressions e whenever $e \Rightarrow_{\mathrm{v}} v$ and $\vdash e : \tau$, then $\vdash v : \tau$. Similarly the call-by-name semantics is type sound if for all closed expressions e whenever $e \Rightarrow_{\mathrm{n}} v$ and $\vdash e : \tau$, then $\vdash v : \tau$. Again it is not difficult to show, by induction on the structure of evaluation, that both operational semantics given are type sound if one assumes that the unspecified rules for constant application are also type sound.

2.4 The Denotational Semantics

The operational semantics given in the previous section is intuitive and easy to understand but not compositional. For a semantics to be compositional the behavior of an expression must depend only on the behavior of its subparts. This property holds for some rules of the operational semantics (such as conditionals), but not in others (such as application or recursion). Compositionality is desirable because we know that the meaning of an expression e is dependent only on the subparts of e. If we add new constants or new language constructs, we automatically know that the meaning of e is unaffected.

The primary difficulty in defining a compositional semantics lies in the definition of recursion. The problem is that the behavior of a recursive expression is inherently self-referential; for example, one possible definition for the expression `rec` $x.e$ might be

$$\mathcal{M}[\![\texttt{rec } x.e]\!]\rho = \mathcal{M}[\![e]\!]\rho[x \mapsto \mathcal{M}[\![\texttt{rec } x.e]\!]\rho] \tag{2.1}$$

where ρ maps variables to semantic meanings and $\rho[x \mapsto d]$ equals the partial function ρ' where $\rho'(x) = d$ and $\rho'(y) = \rho(y)$ for $y \neq x$.

The problem with this equation is that $\mathcal{M}[\![\texttt{rec } x.e]\!]$ appears on both sides of the equation. Therefore it is necessary to ensure that such an equation has a solution and, if there is more than one solution, to specify which one is intended. Furthermore, if an expression does not (operationally) terminate but is otherwise well formed, we still need to give it a meaning; it may be part of an expression that does terminate. For example, `rec` $z.z$ does not terminate in that there is no value v such that `rec` $z.z \Rightarrow_{\mathrm{v}} v$ or `rec` $z.z \Rightarrow_{\mathrm{n}} v$. The expression `lam` $x.$`rec` $z.z$, however, does terminate under both evaluation strategies, evaluating to itself.

2.4.1 Domains and least fixed points

The conventional way (see [38]) to obtain a compositional semantics is through the use of domain theory and *fixed points*. For example, to find the meaning of `rec` $x.e$, we would first define a function F as follows:

$$F(f) = \mathcal{M}[\![e]\!]\rho[x \mapsto f]$$

Any fixed point of f has the property that $F(f) = f$, that is, it is a potentially valid definition of $\mathcal{M}[\![\texttt{rec } x.e]\!]$ in equation 2.1. Furthermore, if F is function from a set D to itself, the following conditions guarantee the existence of a fixed point:

- D is *partially ordered*
- D is *strict*, that is, D has a least element, denoted $\perp$ or $\perp_D$
- D is *ω-complete*, that is, for every chain of elements $d_1 \leq d_2 \leq \ldots$ in D, there exists a least upper bound in D, denoted $\bigsqcup_{i=1}^{\infty} d_i$
- F is *ω-continuous*, that is, F preserves both the ordering on D and the limits of chains

A set D, equipped with the above requirements, is called an *ω-complete partial order*. Because it is also part of one of the more common models of the fixed-point semantics, such sets are also frequently what is meant by the term *domain*. In this dissertation, a *domain* is precisely an ω-complete partial order.

We can find a fixed point by taking the least upper bound of the sequence $\perp \leq F(\perp) \leq \ldots$ This fixed point also has the property of being the *least* fixed point under the ordering in D.

For the particular expression `rec` $x.x$, the function F becomes the identity function; as all elements of D are fixed points the least fixed point is $\perp$ (as can also be seen by taking the limit of the chain $F^{(n)}(\perp) = \perp$). Operationally, as noted earlier, `rec` $x.x$ does not evaluate to any value. Furthermore, as `rec` $x.x$ can have any type we know that for each type there is a non-terminating expression, and its meaning is $\perp$. We show in section 2.6 that $\perp$ is precisely the meaning of (closed, well-typed) non-terminating expressions.

Another way to think about the order used for domains is that it relates to the amount of information that is known about the related expression. There is no useful information that can be known about `rec` $x.x$ and any attempt to evaluate the expression leads to non-termination. Thus its meaning, $\perp$, is the least element in the domain. On the other hand, we know everything about the meanings of `true` and `false` and they can be fully evaluated safely, so their meanings will be greater than $\perp$ and unrelated to each other. For an intermediate example, we can safely determine that `true`::`rec` $x.x$ is not `nil`, and we can evaluate its head, but not its tail. Therefore we would expect that $\perp$ is less than (or equal to) the meaning of `true`::`rec` $x.x$ which is less than the meaning of `true`::`nil`. For strict lists, evaluating any part of a list means fully evaluating the list, so the meaning of `true`::`rec` $x.x$ would be $\perp$; however, for lazy lists evaluation stops once we know that the list involves a `cons`, so we would expect that its meaning would be something other than $\perp$.

2.4.2 Extensional semantics with domains

Unlike the operational semantics, the denotational semantics is not defined solely on closed expressions; one cannot always define the meaning of a closed expression compositionally without needing the meaning of non-closed expressions because there are closed expressions (such as abstractions) that contain non-closed subexpressions. Therefore the meaning of an expression depends not only on the expression itself but also on the meanings assigned to its free variables. We additionally add types both to remove from consideration expressions with type errors (as we did for the operational semantics) and to simplify the domains used. Thus when we talk about the meaning of an expression e we are actually talking about the meaning of the judgement that e has type τ given type environment Γ, written as $\Gamma \vdash e : \tau$. This meaning will be a function from environments consistent with Γ to a domain associated with type τ.

Before defining the function itself, we need to define an environment for valid type assignments Γ and an interpretation for each type τ. For each τ, let $\mathcal{T}[\![\tau]\!]$ be a domain. In this section we only concentrate on the types that will always be present; in section 2.7 we examine the meanings of possible additional types. For the boolean type let $\mathcal{T}[\![\mathbf{bool}]\!]$ be the domain $\mathbf{B}_\perp$ consisting of the elements $\{\perp, \mathrm{tt}, \mathrm{ff}\}$, where $\perp$ is the least element and tt and ff are unrelated to each other. For functional types, the interpretation will differ depending on whether we are considering the call-by-name evaluation strategy or the call-by-value evaluation strategy. For call-by-value, any expression of functional type will not terminate if applied to a non-terminating expression; this property does not necessarily hold for call-by-name. A function f from one domain to another is *strict* if $f(\perp) = \perp$. For the call-by-name semantics the meaning of $\tau_1 \to \tau_2$ is the domain of ω-continuous functions from $\mathcal{T}[\![\tau_1]\!]$ to $\mathcal{T}[\![\tau_2]\!]$, written as $[\mathcal{T}[\![\tau_1]\!] \Rightarrow \mathcal{T}[\![\tau_2]\!]]$. For the call-by-value

	Extensional with domains	Extensional with categories	Intensional
Call-by-value:			
types	$\mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau]\!]$	$\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$	$\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$
environments	$\mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\Gamma]\!]$	$\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\Gamma]\!]$	$\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]$
expressions	$\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e : \tau]\!]$	$\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!]$	$\mathcal{V}[\![\Gamma \vdash e : \tau]\!]$
constants	$\mathcal{C}_{\mathrm{D}}^{\mathrm{V}}[\![c]\!]$	$\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![c]\!]$	$\mathcal{C}^{\mathrm{V}}[\![c]\!]$
Call-by-name:			
types	$\mathcal{T}_{\mathrm{D}}^{\mathrm{N}}[\![\tau]\!]$	$\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau]\!]$	$\mathcal{T}^{\mathrm{N}}[\![\tau]\!]$
environments	$\mathcal{T}_{\mathrm{D}}^{\mathrm{N}}[\![\Gamma]\!]$	$\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\Gamma]\!]$	$\mathcal{T}^{\mathrm{N}}[\![\Gamma]\!]$
expressions	$\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e : \tau]\!]$	$\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!]$	$\mathcal{N}[\![\Gamma \vdash e : \tau]\!]$
constants	$\mathcal{C}_{\mathrm{D}}^{\mathrm{N}}[\![c]\!]$	$\mathcal{C}_{\mathrm{E}}^{\mathrm{N}}[\![c]\!]$	$\mathcal{C}^{\mathrm{N}}[\![c]\!]$

Table 2.2: Semantic function nomenclature

semantics the meaning of $\tau_1 \to \tau_2$ is the domain of *strict* ω-continuous functions from $\mathcal{T}[\![\tau_1]\!]$ to $\mathcal{T}[\![\tau_2]\!]$, written as $[\mathcal{T}[\![\tau_1]\!] \Rightarrow^\circ \mathcal{T}[\![\tau_2]\!]]$.

There are several ways to define environments; for our purposes we have chosen to state that given a type environment Γ, $[\![\Gamma]\!]$ is the set of environments consistent with Γ, where an environment is a partial function from variables to a collection of domains and an environment ρ is consistent with Γ if for each $x : \tau$ in Γ, $\rho(x)$ is in $\mathcal{T}[\![\tau]\!]$.

Remark: So far we have used the same function name to describe the semantics for both the call-by-name and call-by-value evaluation strategies. As they are not the same function, it would be more accurate to use different names. Similarly, we will want to distinguish between the extensional semantics defined in this chapter from the intensional semantics defined in the rest of the dissertation, and between the semantic functions defined in this section and the functions defined in section 2.4.6 (using category theory). Table 2.2 lists the function names to be used throughout the dissertation.

One of the more common definitions used for a denotational semantics (for example, see [11]) is sufficient whenever all one is interested in is the behavior of expressions of ground type. There is, however, a problem: There are terminating expressions (not of ground type) whose meanings are $\perp$ (such as `lam` x.`rec` $z.z$). This problem occurs because we are not distinguishing between a terminating expression of functional type that never returns when applied and an expression of functional type that never terminates in the first place. In situations where the only values of interest are of ground types, it may be appropriate not to distinguish between the two; for example, for the call-by-name, any expression of ground type containing `lam` x.`rec` $z.z$ can have that part of the expression replaced with `rec` $z.z$ without affecting the final result. For this dissertation, however, we will be linking non-termination with intensional cost, so we require that $\perp$ precisely represents the non-terminating expressions of all types, not just of ground types.

The solution to the problem is to *lift* the meaning of function types. Given a domain D, its lift, $D_\perp$ is the domain D with an additional least element added, i.e., $D_\perp = D \cup \{\perp\}$. We originally defined the meaning of a boolean type as a lifted domain; we could have also defined the pre-order $\mathbf{B}$ as the discretely ordered set $\{\mathrm{tt}, \mathrm{ff}\}$. Then $\mathbf{B}_\perp$ is the same domain we previously defined.

With lifts, let

$$\mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau_1 \to \tau_2]\!] = [\mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau_1]\!] \Rightarrow^\circ \mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau_2]\!]]_\perp \qquad \text{and} \qquad \mathcal{T}_{\mathrm{D}}^{\mathrm{N}}[\![\tau_1 \to \tau_2]\!] = [\mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau_1]\!] \Rightarrow \mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau_2]\!]]_\perp$$

$$
\begin{array}{lcl}
\mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau_1 \to \tau_2]\!] & = & [\mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau_1]\!] \Rightarrow^{\circ} \mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau_2]\!]]_{\perp} \\
\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash x : \tau]\!]\rho & = & \rho(x) \\
\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash c : \tau]\!]\rho & = & \mathcal{C}_{\mathrm{D}}^{\mathrm{V}}[\![c : \tau]\!] \\
\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau]\!]\rho & = & \begin{cases} \mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_2 : \tau]\!]\rho & \text{if } \mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!]\rho = \mathrm{tt} \\ \mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_3 : \tau]\!]\rho & \text{if } \mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!]\rho = \mathrm{ff} \\ \perp & \text{otherwise} \end{cases} \\
\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash \texttt{lam } x.e : \tau' \to \tau]\!]\rho & = & \mathsf{up}(\mathsf{strict}(\lambda d.\mathcal{V}_{\mathrm{D}}[\![\Gamma, x : \tau' \vdash e : \tau]\!]\rho[x \mapsto d])) \\
\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_1(e_2) : \tau]\!]\rho & = & (\mathsf{down}(\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!]\rho))(\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_2 : \tau']\!]\rho) \\
\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash \texttt{rec } x.e : \tau]\!]\rho & = & \mathrm{fix}(\lambda d.\mathcal{V}_{\mathrm{D}}[\![\Gamma, x : \tau \vdash e : \tau]\!](\rho[x \mapsto d]))
\end{array}
$$

Call-by-value

$$
\begin{array}{lcl}
\mathcal{T}_{\mathrm{D}}^{\mathrm{N}}[\![\tau_1 \to \tau_2]\!] & = & [\mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau_1]\!] \Rightarrow \mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau_2]\!]]_{\perp} \\
\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash x : \tau]\!]\rho & = & \rho(x) \\
\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash c : \tau]\!]\rho & = & \mathcal{C}_{\mathrm{D}}^{\mathrm{N}}[\![c : \tau]\!] \\
\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau]\!]\rho & = & \begin{cases} \mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_2 : \tau]\!]\rho & \text{if } \mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!]\rho = \mathrm{tt} \\ \mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_3 : \tau]\!]\rho & \text{if } \mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!]\rho = \mathrm{ff} \\ \perp & \text{otherwise} \end{cases} \\
\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash \texttt{lam } x.e : \tau' \to \tau]\!]\rho & = & \mathsf{up}(\lambda d.\mathcal{N}_{\mathrm{D}}[\![\Gamma, x : \tau' \vdash e : \tau]\!]\rho[x \mapsto d]) \\
\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_1(e_2) : \tau]\!]\rho & = & (\mathsf{down}(\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!]\rho))(\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_2 : \tau']\!]\rho) \\
\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash \texttt{rec } x.e : \tau]\!]\rho & = & \mathrm{fix}(\lambda d.\mathcal{N}_{\mathrm{D}}[\![\Gamma, x : \tau \vdash e : \tau]\!](\rho[x \mapsto d]))
\end{array}
$$

Call-by-name

Figure 2.8: Denotational semantics using domains for call-by-value and call-by-name

For any domain D let the function $\mathsf{up} : D \to D_{\perp}$ be the inclusion function and let the function $\mathsf{down} : D_{\perp} \to D$ be the function such that $\mathsf{down}(\perp)$ is the least element of D and for all $x \in D$, $\mathsf{down}(\mathsf{up}(x)) = x$. Also, if $f_{\perp}$ is a function from $D_{\perp}$ to $D'_{\perp}$, let $\mathsf{strict}(f_{\perp})$ be the function such that $\mathsf{strict}(f_{\perp})(\perp) = \perp$ and $\mathsf{strict}(f_{\perp})(\mathsf{up}(x)) = f_{\perp}(\mathsf{up}(x))$; that is, $\mathsf{strict}(f_{\perp})$ is the strict version of $f_{\perp}$. Figure 2.8 contains the adjusted semantics.

Assuming that the definition of the constants properly matches the operational semantics, we have the following soundness properties:

Theorem 2.4.1 *For any closed expression e of type τ,*

1. *If $e \Rightarrow_{\mathrm{v}} v$ then $\mathcal{V}_{\mathrm{D}}[\![\vdash e : \tau]\!] = \mathcal{V}_{\mathrm{D}}[\![\vdash v : \tau]\!]$*
2. *If $e \Rightarrow_{\mathrm{n}} v$ then $\mathcal{N}_{\mathrm{D}}[\![\vdash e : \tau]\!] = \mathcal{N}_{\mathrm{D}}[\![\vdash v : \tau]\!]$*

Proof. The proof can be found in a number of sources, including [11]. □

A semantic definition if *adequate* if $\perp$ solely represents non-termination. We have the following adequacy properties:

Theorem 2.4.2 *For any closed expression e of type τ,*

1. *$\mathcal{V}_{\mathrm{D}}[\![\vdash e : \tau]\!] \neq \perp$ if and only if for some value v, $e \Rightarrow_{\mathrm{v}} v$, and*

2. $\mathcal{N}_{\mathrm{D}}[\![\vdash e : \tau]\!] \neq \bot$ *if and only if for some value* v, $e \Rightarrow_{\mathrm{n}} v$.

Proof. The proof can be found in a number of sources, including [11]. □

2.4.3 Categories

While domains are sufficient for most forms of analysis, for our purposes we would rather use a more abstract representation, in this case category theory. By using category theory we can not only better separate the algebraic properties we desire from those that are inherently part of a domain-based model, but we can also define and prove many properties with the domain model in mind and still have the proven properties available when we eventually shift to a non domain-based model.

There are many publications explaining the basic categorical concepts used in this thesis. Pierce's book [31] is a good introduction, [3] is aimed for computer scientists, while [23], written for mathematicians, is often cited in the literature.

Formal Definitions

A *category* $\mathcal{C}$ consists of three parts:

- A collection of *objects*
- A collection of *morphisms*, $\mathrm{Hom}(A, B)$, for each pair of objects A and B. A morphism in $\mathrm{Hom}(A, B)$ can be described either as a morphism *from* A *to* B or as a morphism $f : A \to B$. If $f : A \to B$, then A is said to be its *domain* and B is its *codomain*.
- For each triple of objects A, B, C, a composition operator $\circ$: that combines a morphism in $\mathrm{Hom}(A, B)$ and a morphism in $\mathrm{Hom}(B, C)$ into a morphism in $\mathrm{Hom}(A, C)$. For example, if $f : A \to B$ and $g : B \to C$, then $g \circ f : A \to C$.

These parts must satisfy the following requirements:

1. Existence of identity morphisms: For each object A there exists a morphism $\mathsf{id}_A : A \to A$ such that for any $f : A \to B$ and $g : C \to A$, $f \circ \mathsf{id}_A = f$ and $\mathsf{id}_A \circ g = g$.
2. Associativity of composition: If $f : A \to B$, $g : B \to C$, and $h : C \to D$, then

$$h \circ (g \circ f) = (h \circ g) \circ f$$

There are a number of categories where the objects are sets (with possibly some added structure) and the morphisms are (structure-preserving) functions. The simplest one is **Set**; an object in **Set** is a set, and a morphism from A to B in **Set** is a (total) function from A to B; composition in **Set** is standard function composition. Similarly, domains form a category, **Dom**, where objects are domains, morphisms are ω-continuous functions and composition is also the standard notion of function composition. Other categories include the category of groups and homomorphisms and the category of partial orders and monotone functions.

There are also categories whose morphisms are not functions. Any two categories $\mathcal{C}_1$ and $\mathcal{C}_2$ can be used to form a *product category* $\mathcal{C}_1 \times \mathcal{C}_2$. An object in $\mathcal{C}_1 \times \mathcal{C}_2$ is a pair of objects (A_1, A_2) from $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, and a morphism $(f, g) : (A_1, A_2) \to (B_1, B_2)$ in $\mathcal{C}_1 \times \mathcal{C}_2$ is a pair of morphisms $f : A_1 \to B_1$ in $\mathcal{C}_1$ and $g : A_2 \to B_2$ in $\mathcal{C}_2$. Composition is performed pairwise; $(f, g) \circ (f', g') = (f \circ f', g \circ g')$, and the identity morphism on an object (A_1, A_2) is $(\mathsf{id}_{A_1}, \mathsf{id}_{A_2})$.

Similarly, given a category $\mathcal{C}$ we can form another category $\mathcal{C}^{\mathrm{op}}$, called the *opposite category*. The objects of $\mathcal{C}^{\mathrm{op}}$ are the objects of $\mathcal{C}$, but the morphisms are reversed, that is, f is a morphism from A to B in $\mathcal{C}^{\mathrm{op}}$ exactly when f is a morphism from B to A in $\mathcal{C}$.

The category of domains gives us a starting point for determining a denotational semantics using categories. Ideally we would want the denotational semantics with domains to be a special case of the denotational semantics with categories (or at least equivalent to a special case). For the denotational semantics with domains, the meanings of types are domains and the meaning of (typed) expressions are continuous functions; therefore, we expect that the meaning of types in the categorical semantics will be objects while the meaning of expressions will be morphisms.

Some categories have objects and morphisms with special "universality" properties. An object in a category $\mathcal{C}$ is *terminal* if for any object A there exists exactly one morphism from A to the terminal object. If a category has a terminal object we usually write $\mathbf{1}$ to denote some distinguished terminal object; the unique morphism from A to $\mathbf{1}$ is then denoted $!_A$.

Suppose that $f : A \to B$ and $g : B \to A$. If $g \circ f = \mathsf{id}_A$ and $f \circ g = \mathsf{id}_B$ then A and B are *isomorphic* with f and g being *isomorphisms*. Many categorical definitions are only unique "up to isomorphism." For example, the definition of a terminal object does not uniquely determine an object, but all terminal objects in a given category are isomorphic.

Comparisons between categories are often more important than particular categories or morphisms. A *functor* F from a category $\mathcal{C}$ to a category $\mathcal{D}$ (written $F : \mathcal{C} \to \mathcal{D}$) consists of a function (also written as F) from objects of $\mathcal{C}$ to objects of $\mathcal{D}$ and a function (again written as F) on morphisms where for each morphism $f : A \to B$ in $\mathcal{C}$, Ff is a morphism in $\mathcal{D}$ from FA to FB. Furthermore for each object A of $\mathcal{C}$, $F\mathsf{id}_A = \mathsf{id}_{FA}$, and for each pair of morphisms $f : A \to B$ and $g : B \to C$ in $\mathcal{C}$, $F(g \circ f) = Fg \circ Ff$.

A special kind of functor is a *forgetful* functor. Forgetful functors generally go from a category of structured objects to a category of similar objects with less structure. For example, there is a forgetful functor from **Dom** to **Set**; that functor simply "forgets" the ordering of the domains and extracts the underlying set.

For any object B in $\mathcal{C}$ and any category $\mathcal{D}$, there is a degenerate "constant" functor B from $\mathcal{D}$ to $\mathcal{C}$, where, for all objects A in $\mathcal{D}$, $BA = B$ and, for all morphisms $f : A \to B$, $Bf = \mathsf{id}_B$. Another simple functor of note is the identity functor I from a category to itself, which maps objects and morphisms to themselves. In **Dom**, let $LA = A_\perp$ and for $f : A \to B$, let $Lf : LA \to LB$ be the function such that $Lf(\perp) = \perp$ and $Lf(\mathsf{up}(a)) = \mathsf{up}(f(a))$. Then L is a functor, called the lifting functor, from **Dom** to itself.

Given two functors $F, G : \mathcal{C} \to \mathcal{D}$, a *natural transformation* $\alpha : F \stackrel{.}{\to} G$, is a collection of morphisms such that for each object A in $\mathcal{C}$, $\alpha_A : FA \to GA$ in $\mathcal{D}$ and for each morphism $f : A \to B$ in $\mathcal{C}$, $\alpha_B \circ Ff = Gf \circ \alpha_A$. A *natural isomorphism* is a natural transformation α such that for each object A, α_A is an isomorphism. For any category with a terminal element $\mathbf{1}$, $!$ is a natural transformation from the identity functor I to the degenerate functor $\mathbf{1}$. Also, in **Dom**, up is a natural transformation from I to L.

A functor of two arguments from categories $\mathcal{C}_1$ and $\mathcal{C}_2$ is a functor (of one argument) from $\mathcal{C}_1 \times \mathcal{C}_2$. A functor F from $\mathcal{C}$ to $\mathcal{D}$ is *contravariant* if it satisfies the above requirements except that $F(f \circ g) = Fg \circ Ff$; formally a contravariant functor is just an ordinary functor from the opposite category $\mathcal{C}^{\mathrm{op}}$ to $\mathcal{D}$.

Category theorists often use *diagrams* to display equations. A diagram is a directed graph where the nodes are objects and the edges are morphisms. Such a diagram *commutes* if all paths between a pair of objects consisting of a least two edges correspond to equal morphisms. It is common in the category theoretic setting to use commuting diagrams to represent or display equations; for

$$\begin{tikzcd} FA \arrow[r, "Ff"] \arrow[d, "\alpha_A"'] & FB \arrow[d, "\alpha_B"] \\ GA \arrow[r, "Gf"'] & GB \end{tikzcd}$$

Figure 2.9: Naturality requirement

example, the diagram in Figure 2.9 represents the condition required for naturality.

In our semantics we will use functors and natural transformations in a number of ways, but one use is worth noting here. The semantic functions given for domains do not include any constructed data types. The semantics given for categories will specify that constructed data types will be formed using functors. While we only need to know how the functors act on objects in order to show soundness and adequacy overall, the functors' actions on morphisms help convince us that the data types are well formed and provide the structure needed to prove soundness of particular constants.

Cartesian closed categories

One of the most important discoveries made concerning semantics and categories is that the semantics of the simply typed λ-calculus is closely aligned with Cartesian closed categories ([19]). Even though our language differs from the simply typed λ-calculus in that we include different evaluation strategies plus elements such as conditionals, Cartesian closed categories are invaluable conceptually.

A *product* of two objects A and B is an object, written as $A \times B$, together with two projection morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ such that for all morphisms $f : C \to A$ and $g : C \to B$, there is a unique morphism $\langle f, g\rangle$ from C to $A \times B$ satisfying $\pi_1 \circ \langle f, g\rangle = f$ and $\pi_2 \circ \langle f, g\rangle = g$. For all morphisms $f : A \to C$ and $g : B \to D$, we define $f \times g : A \times B \to C \times D$ to be $\langle f \circ \pi_1, g \circ \pi_2\rangle$. A category has *binary products* if every pair of objects has a product.

There may be more than one way to form products from a given pair of objects. In particular, both $A \times B$ and $B \times A$ are products of A and B. All products derived from the same objects, however, are isomorphic to each other. Thus $A \times B$ is isomorphic to $B \times A$ and $A \times (B \times C)$ is isomorphic to $(A \times B) \times C$. Furthermore, if $\mathbf{1}$ is a terminal object, we can easily show that $A \times \mathbf{1}$ is isomorphic to A. Some (natural) product isomorphisms are of frequent use in this dissertation; these are listed in Figure 2.10. Note that α^r and α^l are mutual inverses, while β and ϕ are their own inverses (relative to their subscripts, i.e., $\beta_{A,B}$ and $\beta_{B,A}$ are inverses).

With products and terminal elements we can define n-ary products for all n. While all definitions of n-ary products are isomorphic, for precision we use the following, also defined in [11]:

Definition 2.4.1 Given a category $\mathcal{C}$ with finite products and n objects $A_1, \ldots, A_n$, let the product $\times(A_1, \ldots, A_n)$ be defined inductively as follows:

- $\times() = \mathbf{1}$
- $\times(A_1, \ldots, A_n) = \times(A_1, \ldots, A_{n-1}) \times A_n$

$$
\begin{array}{rlcl}
\beta_{A,B}\colon & A \times B \to B \times A & = & \langle \pi_2, \pi_1 \rangle \\
\alpha^r_{A,B,C}\colon & (A \times B) \times C \to A \times (B \times C) & = & \langle \pi_1 \circ \pi_1, \pi_2 \times \mathsf{id} \rangle \\
\alpha^l_{A,B,C}\colon & A \times (B \times C) \to (A \times B) \times C & = & \langle \mathsf{id} \times \pi_1, \pi_2 \circ \pi_2 \rangle \\
\phi_{A,B,C,D}\colon & (A \times B) \times (C \times D) \to (A \times C) \times (B \times D) & = & \langle \pi_1 \times \pi_1, \pi_2 \times \pi_2 \rangle
\end{array}
$$

Figure 2.10: Commonly used product isomorphisms

Given $\times(A_1, \ldots, A_n)$ and $1 \leq i \leq n$, let π^n_i be the i'th projection, i.e. $\pi^n_n = \pi_2$ and for $i < n$, $\pi^n_i = \pi^{n-1}_i \circ \pi_1$.

Furthermore, if for each $1 \leq i \leq n$, $f_i : C \to A_i$, let $\langle\rangle(f_1, \ldots, f_n) : C \to \times(A_1, \ldots, A_n)$ be defined inductively as follows:

- $\langle\rangle() = !_C$, and for $n > 0$,
- $\langle\rangle(f_1, \ldots, f_n) = \langle \langle\rangle(f_1, \ldots, f_{n-1}), f_n \rangle$

An *exponentiation* for a pair of objects A and B is an object $[A \Rightarrow B]$ and a morphism $\mathsf{app}_{A,B}$ from $[A \Rightarrow B] \times A$ to B such that for all objects C and all morphisms $f : C \times A \to B$ there exists a unique morphism $\mathsf{curry}(f) : C \to [A \Rightarrow B]$ such that $f = \mathsf{app}_{A,B} \circ (\mathsf{curry}(f) \times \mathsf{id}_A)$. Given a morphism $g : C \to [A \to B]$ we define $\mathsf{uncurry}(g) : C \times A \to B$ to be $\mathsf{app}_{A,B} \circ (g \times \mathsf{id}_A)$.

A category is *Cartesian closed* if it has binary products, a terminal object, and every pair of objects A and B has an exponentiation. It follows that the category has finite products.

Both **Set** and **Dom** are Cartesian closed. For **Set**, the product object $A \times B$ is the set of pairs from A and B, the exponentiation object $[A \Rightarrow B]$ is the set of functions from A to B, and app performs standard function application. For **Dom**, the product object $A \times B$ is the domain of pairs from A and B, ordered pairwise, the exponentiation object $[A \Rightarrow B]$ is the domain of ω-continuous functions, ordered element-wise, and app is again standard function application. Exponentiation models hom-sets within the category; note that in both cases the exponentiation object is the hom-set itself (with ordering added, in the case of **Dom**).

Both products and exponentiation produce bifunctors: the product functor, $- \times -$ from $\mathcal{C} \times \mathcal{C}$ to $\mathcal{C}$ and the exponentiation functor $[- \Rightarrow -]$ from $\mathcal{C}^{\mathrm{op}} \times \mathcal{C}$ to $\mathcal{C}$. Given $f : A \to B$ and $g : C \to D$,

$$f \times g = \langle f \circ \pi_1, g \circ \pi_2 \rangle : A \times C \to B \times D$$

and we define

$$[f \Rightarrow g] = \mathsf{curry}(g \circ \mathsf{app} \circ (\mathsf{id} \times f)) : [B \Rightarrow C] \to [A \Rightarrow D]$$

If $\mathcal{C}$ is a Cartesian closed category, then $\mathrm{Hom}(C, A \times B)$ is isomorphic to $\mathrm{Hom}(C, A) \times \mathrm{Hom}(C, B)$ (in **Set**) and $\mathrm{Hom}(C \times A, B)$ is isomorphic to $\mathrm{Hom}(C, [A \Rightarrow B])$. These isomorphisms are needed for the definition of enriched categories in section 2.4.3.

There are many uses for products: aside from modeling products in the language itself we will need them to form environments. We need exponentiation to model function types.

Other constructions

A category has *binary coproducts* if for every pair of objects A and B, there is an object $A + B$ and a pair of injection morphisms $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$, such that for all morphisms

$f : A \to C$ and $g : B \to C$, there is a unique morphism $[f, g]$ from $A+B$ to C satisfying $[f, g] \circ \mathsf{inl} = f$ and $[f, g] \circ \mathsf{inr} = g$. For all morphisms $f : A \to C$ and $g : B \to D$, we define $f + g : A + B \to C + D$ to be $[\mathsf{inl} \circ f, \mathsf{inr} \circ g]$. Thus $- + -$ is also a functor from $\mathcal{C} \times \mathcal{C}$ to $\mathcal{C}$.

Set has coproducts; the coproduct of A and B is their disjoint union. **Dom** does not have coproducts; the disjoint union of A and B does not have a least element. By either adding a least element (resulting in a *separated sum*) or sharing the least elements (result in a *smashed sum*) we get an object with some but not all of the properties of coproducts. For example, with smashed sums the construction $[f, g]$ only satisfies the required equations when f and g are both strict. For separated sums there is no *unique* construction $[f, g]$ satisfying the other requirements. As coproducts are needed to construct sums and lists, we wanted a way to have coproducts in our categories. It is not possible to have coproducts, Cartesian closure, and still guarantee that all morphisms on a single domain have fixed points. The only category satisfying all those requirements is the trivial category with one element and one morphism. [15]

In domain theory, the traditional solution to this problem is to relax the requirement that all objects have coproducts. Another common solution, however, is to relax the requirement for fixed points, and we choose to use that solution in this dissertation. Thus instead of the category **Dom** we instead use **PDom**, the category of *pre-domains* (and continuous functions), that is, ω-complete partial orders that do not need to have a least element. This category is Cartesian closed and has coproducts. It is not true, however, that for all D and all $f : D \to D$, f will have a fixed point; for example, the function $\mathsf{not} : \mathbf{B} \to \mathbf{B}$ defined as $\mathsf{not}(\mathrm{tt}) = \mathrm{ff}$ and $\mathsf{not}(\mathrm{ff}) = \mathrm{tt}$ does not have a fixed point. If, however, a pre-domain D is known to have a least element (i.e., is actually a domain), then we can still guarantee that a continuous function $f : D \to D$ has a (least) fixed point. Therefore to use this category we must ensure that we only calculate fixed points on string pre-domains.

One extra advantage we obtain with **PDom** is that we can use the L functor to convert pre-domains to domains. We can then generalize the process to any category that has a functor L satisfying certain properties, as described in section 2.4.5. Thus we not only can use other categories besides **PDom**, but the semantics that follows from this treatment explicitly models the behavior of non-termination. As properties of non-termination are closely related to properties of cost accounting, this enables us to better model the behavior of costs.

Category theory and fixed points

While a Cartesian closed category is sufficient to model the λ-calculus, it is no longer sufficient once we add fixed points. The fixed point construction in [3] states that when the hom-sets are domains (i.e., there is a strict, ω-complete partial ordering on the morphisms) then given a function $f : D \to D$, there is a morphism $\mathsf{fix}(f) : \mathbf{1} \to D$ such that $f \circ \mathsf{fix}(f) = \mathsf{fix}(f)$. For this dissertation, however, we would prefer instead to convert a morphism $f : A \times D \to D$ to a morphism from A to D instead, where A represents the environment. On the basis of constructions from other sources ([49], [28]) we can construct a similar but more directly useful fixed point operator.

Definition 2.4.2 A category $\mathcal{C}$ is **PDom**-*enriched* if each hom-set is (also) an element of **PDom** and for any objects A, B, and C of $\mathcal{C}$, composition is (also) a morphism in **PDom** from the product $\mathrm{Hom}(A, B) \times \mathrm{Hom}(B, C)$ to $\mathrm{Hom}(A, C)$, i.e., composition is ω-continuous.

The "(also)" in the above definition reminds us that the hom-sets are also sets and composition is also a function, so $\mathcal{C}$ is also a category in the usual sense.

Any category can be considered trivially **PDom**-enriched by using the discrete ordering on all of the hom-sets. A non-trivial example of an enriched category is **PDom** itself, where the ordering on the hom-sets is the usual pointwise ordering. Also, **Dom** is **PDom** enriched, with the added property that all orderings are strict.

A **PDom**-enriched category is Cartesian closed if it is Cartesian closed in the usual sense, and the isomorphisms on the hom-sets generated by Cartesian closure (noted in the previous section) are morphisms in **PDom**, that is, they are ω-continuous functions. In this way currying and product formation preserve the orderings of the hom-sets in the category. Thus the two isomorphisms between $\mathrm{Hom}(C \times A, B)$ and $\mathrm{Hom}(C, [A \Rightarrow B])$, namely curry and $\mathsf{uncurry}$, are continuous. Therefore, for any $f, f' : C \to [A \Rightarrow B]$, $f \leq f'$ if and only if $\mathsf{app} \circ (f \times \mathsf{id}_A) \leq \mathsf{app} \circ (f' \times \mathsf{id}_A)$.

Similarly, for products the isomorphisms are $\phi(h) = (\pi_1 \circ h, \pi_2 \circ h)$ and $\phi^{-1}(f, g) = \langle f, g\rangle$, where (f, g) indicates a pair of morphisms in **Set** (or **PDom**). Thus if $f, f' : C \to A$ and $g, g' : C \to B$, then $\langle f, g\rangle \leq \langle f', g'\rangle$ if and only if

$$\phi\langle f, g\rangle = (\pi_1 \circ \langle f, g\rangle, \pi_2 \circ \langle f, g\rangle) = (f, g) \leq \phi\langle f', g'\rangle = (f', g')$$

i.e., if and only if $f \leq f'$ and $g \leq g'$. Also, since ϕ and ϕ' are continuous, if $f_0 \leq f_1 \leq \ldots$ is a chain of morphisms from C to A and $g_0 \leq g_1 \leq \ldots$ is a chain of morphisms from C to B, then

$$\bigsqcup_{n=0}^{\infty} \langle f_n, g_n\rangle = \langle \bigsqcup_{n=0}^{\infty} f_n, \bigsqcup_{n=0}^{\infty} g_n\rangle$$

Furthermore, by the continuity of composition,

$$(\bigsqcup_{n=0}^{\infty} g_n) \circ (\bigsqcup_{n=0}^{\infty} f_n) = \bigsqcup_{n=0}^{\infty} (g_n \circ f_n)$$

as well.

For fixed points we need something analogous to $\perp$. Let an object D in $\mathcal{C}$ be *strict* if there exists a morphism $\perp_D : \mathbf{1} \to D$ such that for every $f : A \to D$, $\perp_D \circ !_A \leq_D f$. For strict objects, it is then possible to generate (least) fixed points:

Theorem 2.4.3 *Suppose that D is a strict object of $\mathcal{C}$, and that $f : D' \times D \to D$. Then there exists a least morphism $\mathsf{fixp}_{D,D'}(f) : D' \to D$ such that $f \circ \langle \mathsf{id}_{D'}, \mathsf{fixp}_{D,D'}(f)\rangle = \mathsf{fixp}_{D,D'}(f)$.*

Proof. This is essentially the proof shown in [3] with the extra object D' included.

For $n \geq 0$, let $f_n : D' \to D$ be defined inductively as follows:

$$\begin{aligned} f_0 &= \perp_D \circ !_{D'} \\ f_{n+1} &= f \circ \langle \mathsf{id}_{D'}, f_n\rangle \end{aligned}$$

First we must show that this forms a chain $f_0 \leq f_1 \ldots$. By strictness $f_0 \leq_D f_1$. If $f_{n-1} \leq_D f_n$ then $\langle \mathsf{id}_{D'}, f_{n-1}\rangle \leq_{D' \times D} \langle \mathsf{id}_{D'}, f_n\rangle$ by the component-wise ordering of products, so

$$f_n = f \circ \langle \mathsf{id}_{D'}, f_{n-1}\rangle \leq_D f \circ \langle \mathsf{id}_{D'}, f_n\rangle = f_{n+1}$$

Thus by induction on n, f_n is a chain of morphisms from D' to D.

Now let $\mathsf{fixp}_{D,D'}(f) = \bigsqcup_{n=0}^{\infty} f_n$. Note that by the continuity of products and composition,

$$\begin{aligned} f \circ \langle \mathsf{id}_{D'}, \mathsf{fixp}_{D,D'}(f)\rangle &= f \circ \langle \mathsf{id}_{D'}, \textstyle\bigsqcup_{n=0}^{\infty} f_n\rangle \\ &= \textstyle\bigsqcup_{n=0}^{\infty} (f \circ \langle \mathsf{id}_{D'}, f_n\rangle) \\ &= \textstyle\bigsqcup_{n=0}^{\infty} f_{n+1} \\ &= \mathsf{fixp}_{D,D'}(f) \end{aligned}$$

Let $h : D' \to D$ such that $f \circ \langle \mathsf{id}_{D'}, h\rangle = h$. By strictness $f_0 \leq h$, and if $f_n \leq h$ then

$$f_{n+1} = f \circ \langle \mathsf{id}_{D'}, f_n\rangle \leq f \circ \langle \mathsf{id}_{D'}, h\rangle = h$$

Thus h is an upper bound of the f_n's, so $\mathsf{fixp}_{D,D'}(f) = \bigsqcup_{n=0}^{\infty} f_n \leq h$. $\square$

Alternatively, we can define a collection of morphisms $\mathsf{fix}_D : [D \Rightarrow D] \to D$ as the fixed point operator, with the property that $\mathsf{app} \circ \langle \mathsf{id}_{[D\Rightarrow D]}, \mathsf{fix}_D\rangle = \mathsf{fix}_D$, and fix_D is the least morphism with such a property. The latter is derivable from the former: fix_D is equal to $\mathsf{fixp}_{[D\Rightarrow D],D}(\mathsf{app})$, and if $f : D' \times D \to D$, then $\mathsf{fixp}(f) = \mathsf{fix}_D \circ \mathsf{curry}(f)$. While the latter form more closely resembles the fixed-point constructor used with domains, in this document we use fixp because it both works well with the denotational semantics and does not require the explicit use of application.

One more property of fixp beyond the general fixed point properties is useful. This result is not surprising given the relationship just described between fixp and fix.

Lemma 2.4.4 *Let D_0, D, and D' be objects of $\mathcal{C}$, let f be a morphism from $D \times D'$ to D', and let g be a morphism from D_0 to D. Then*

$$\mathsf{fixp}_{D_0,D'}(f \circ (g \times \mathsf{id}_{D'})) = \mathsf{fixp}_{D,D'}(f) \circ g$$

Proof. For $n \geq 0$, let $f_n : D \to D'$ and $f'_n : D_0 \to D'$ be defined inductively as follows:

$$\begin{aligned} f_0 &= \bot_{D'} \circ !_D & f'_0 &= \bot_{D'} \circ !_{D_0} \\ f_{n+1} &= f \circ \langle \mathsf{id}_D, f_n\rangle & f'_{n+1} &= f \circ (g \times \mathsf{id}_{D'}) \circ \langle \mathsf{id}_D, f'_n\rangle \end{aligned}$$

Thus $\mathsf{fixp}_{D,D'}(f) = \bigsqcup_{n=0}^{\infty} f_n$ and $\mathsf{fixp}_{D_0,D'}(f \circ (g \times \mathsf{id}_{D'})) = \bigsqcup_{n=0}^{\infty} f'_n$.

Now note that $f_0 \circ g = \bot_{D'} \circ !_D \circ g = \bot_{D'} \circ !_{D_0} = f'_0$ by the uniqueness of !. Also if $f'_n = f_n \circ g$ then

$$\begin{aligned} f'_{n+1} &= f \circ (g \times \mathsf{id}_{D'}) \circ \langle \mathsf{id}_D, f'_n\rangle \\ &= f \circ \langle g, f_n \circ g\rangle \\ &= f \circ \langle \mathsf{id}_D, f_n\rangle \circ g \\ &= f_{n+1} \circ g \end{aligned}$$

Thus by induction for all $n \geq 0$, $f'_n = f_n \circ g$ and by the continuity of composition

$$\begin{aligned} \mathsf{fixp}_{D_0,D'}(f \circ (g \times \mathsf{id}_{D'})) &= \textstyle\bigsqcup_{n=0}^{\infty} f'_n \\ &= \textstyle\bigsqcup_{n=0}^{\infty} (f_n \circ g) \\ &= (\textstyle\bigsqcup_{n=0}^{\infty} f_n) \circ g \\ &= \mathsf{fixp}_{D,D'}(f) \circ g \end{aligned}$$

$\square$

2.4.4 Monads

In 1989 Moggi noted that the categorical construct called a *monad* can be used as a general method for modeling certain structures of functional languages ([24]). Since then, researchers have used monads for many purposes, either as part of a semantic definition or as the basis for constructs in functional languages themselves (see [46]). In essence, monads provide a uniform method for handling general procedures such as initializing or combining structures.

There are two different ways to define a monad. The first method uses standard categorical definitions: A *monad* is a functor C along with two natural transformations $\eta : I \dot{\to} C$ and $\mu : C^2 \dot{\to} C$, satisfying the following properties for all objects A:

- $\mu_A \circ \eta_{CA} = \mathsf{id}_{CA}$
- $\mu_A \circ C\eta_A = \mathsf{id}_{CA}$
- $\mu_A \circ C\mu_A = \mu_A \circ \mu_{CA}$.

The other method involves a structure called a *Kleisli triple*. A Kleisli triple is a function C from objects to objects along with, for each object A, a morphism $\eta_A : A \to CA$ and, for each morphism $f : A \to CB$, a morphism $f^* : CA \to CB$, satisfying the following:

- $f^* \circ \eta_A = f$
- ${\eta_A}^* = \mathsf{id}_{CA}$
- For all morphisms $g : B \to CD$, $g^* \circ f^* = (g^* \circ f)^*$

Each Kleisli triple $(C, \eta, (-)^*)$ gives rise to a monad (C, η, μ), where $\mu_A = {\mathsf{id}_{CA}}^*$ and for any morphism $f : A \to B$, $Cf = (\eta_B \circ f)^*$. Similarly, every monad determines a Kleisli triple, where $f^* = \mu_B \circ Cf$. Which version is used, therefore, becomes a matter of personal preference. In this dissertation we will typically use the Kleisli triple functions plus the definition of C on morphisms; we therefore also make frequent use of the following derived properties combining the two forms:

$$Cg \circ f^* = (Cg \circ f)^* \quad \text{and} \quad g^* \circ Cf = (g \circ f)^*$$

A large number of constructions in computer science can be described by a monad. We will make particular use of the monad formed from the lifting functor L along with up and down. The function f^* returns $\perp$ when the input is $\perp$ and applies f otherwise. In **Set** the power set function also forms a monad: If $\mathcal{P}A$ is the powerset of A, then $\eta_A(a) = \{a\}$ and for $f : A \to \mathcal{P}B$, $f^*(X) = \bigcup_{x \in X} f(x)$.

Strong monads

In practice, constructions that can be represented as monads often need some additional morphisms or natural transformations to handle tasks unique to that construction; for example, the lifting monad needs the natural transformation $\perp$ from $\mathbf{1}$ to L to describe non-termination. Many monads, moreover, need some way of interacting with products. In particular, when examining the semantics of high-order functional languages, we need a method for combining a monad with products to enable a valid definition of function application (fortunately, there is no need for any extra properties to handle currying). Category theory already defines a class of monads containing the extra power. A monad is said to be *strong* if there exists a natural transformation τ from $C - \times -$ to $C(- \times -)$ (i.e., $\tau_{A,B} : CA \times B \to C(A \times B)$) satisfying the following four properties:

$$\begin{tikzcd}
& CA \times B \arrow[dl, "\tau"'] \arrow[dr, "\pi_1"] & \\
C(A \times B) \arrow[rr, "C\pi_1"'] & & CA
\end{tikzcd}
\qquad
\begin{tikzcd}
& A \times B \arrow[dl, "\eta \times \mathsf{id}"'] \arrow[dr, "\eta"] & \\
CA \times B \arrow[rr, "\tau"'] & & C(A \times B)
\end{tikzcd}$$

$$\begin{tikzcd}
(CA \times B) \times D \arrow[r, "\tau \times \mathsf{id}"] \arrow[d, "\alpha^r"'] & C(A \times B) \times D \arrow[r, "\tau"] & C((A \times B) \times D) \arrow[d, "C\alpha^r"] \\
CA \times (B \times D) \arrow[rr, "\tau"'] & & C(A \times (B \times D))
\end{tikzcd}
\qquad
\begin{tikzcd}
CA \times B \arrow[r, "f^* \times g"] \arrow[d, "\tau"'] & CA' \times B' \arrow[d, "\tau"] \\
C(A \times B) \arrow[r, "(\tau \circ (f \times g))^*"'] & C(A' \times B')
\end{tikzcd}$$

Figure 2.11: Strength requirements for a monad

- $C\pi_1 \circ \tau_{A,B} = \pi_1$
- $C\alpha^r_{A,B,D} \circ \tau_{A\times B,D} \circ (\tau_{A,B} \times \mathsf{id}_D) = \tau_{A,B\times D} \circ \alpha^r_{CA,B,D}$
- $\tau_{A,B} \circ (\eta_A \times \mathsf{id}_B) = \eta_{A\times B}$
- $\mu_{A\times B} \circ C\tau_{A,B} \circ \tau_{CA,B} = \tau_{A,B} \circ (\mu_A \times \mathsf{id}_B)$

The category where C is defined must obviously have products for this definition to make any sense.

We can also define the above properties using only the Kleisli triple notation by replacing the requirement that τ is natural and the last property with the following (see Figure 2.11):

- $\tau_{A',B'} \circ (f^* \times g) = (\tau_{A',B'} \circ (f \times g))^* \circ \tau_{A,B}$.

The two definitions are equivalent.

There is a another way to denote strength. Rather than use a left-handed natural transformation τ as above we can instead use a natural transformation $\psi_{A,B} : CA \times CB \to C(A \times B)$ satisfying certain commutative diagrams as shown in in the following theorem. This natural transformation "pushes" C outside of a product and does so in a uniform way in that there are no properties that are specific to the left or right side of a product.

Theorem 2.4.5 *Suppose that for each pair of objects A and B, there exists a morphism $\psi_{A,B}$ from $CA \times CB$ to $C(A \times B)$ satisfying the following properties (also see Figure 2.12):*

- $C\pi_1 \circ \psi_{A,B} \circ (\mathsf{id} \times \eta_B) = \pi_1$
- $C\pi_2 \circ \psi_{A,B} \circ (\eta_A \times \mathsf{id}) = \pi_2$

$$
\begin{tikzcd}
CA \times B \arrow[r, "\mathsf{id} \times \eta"] \arrow[d, "\pi_1"'] & CA \times CB \arrow[d, "\psi"] & A \times CB \arrow[l, "\eta \times \mathsf{id}"'] \arrow[d, "\pi_2"] \\
CA & C(A \times B) \arrow[l, "C\pi_1"] \arrow[r, "C\pi_2"'] & CB
\end{tikzcd}
\qquad
\begin{tikzcd}
 & A \times B \arrow[dl, "\eta \times \eta"'] \arrow[dr, "\eta"] & \\
CA \times CB \arrow[rr, "\psi"'] & & C(A \times B)
\end{tikzcd}
$$

$$
\begin{tikzcd}
(CA \times CB) \times CD \arrow[r, "\psi \times \mathsf{id}"] \arrow[d, "\alpha^r"'] & C(A \times B) \times CD \arrow[r, "\psi"] & C((A \times B) \times D) \arrow[d, "C\alpha^r"] \\
CA \times (CB \times CD) \arrow[r, "\mathsf{id} \times \psi"'] & CA \times C(B \times D) \arrow[r, "\psi"'] & C(A \times (B \times D))
\end{tikzcd}
\qquad
\begin{tikzcd}
CA \times CB \arrow[r, "f^* \times g^*"] \arrow[d, "\psi"'] & CA' \times CB' \arrow[d, "\psi"] \\
C(A \times B) \arrow[r, "(\psi \circ (f \times g))^*"'] & C(A' \times B')
\end{tikzcd}
$$

Figure 2.12: Strength properties with ψ

- $\psi_{A,B} \circ (\eta_A \times \eta_B) = \eta_{A \times B}$
- $\psi_{A,B \times D} \circ (\mathsf{id}_{CA} \times \psi_{B,D}) \circ \alpha^r_{CA,CB,CD} = C\alpha^r_{A,B,D} \circ \psi_{A \times B,D} \circ (\psi_{A,B} \times \mathsf{id}_{CD})$
- $\psi_{A',B'} \circ (f^* \times g^*) = (\psi_{A',B'} \circ (f \times g))^* \circ \psi_{A,B}$

Then $(C, \eta, (-)^*, \tau)$ *is a strong monad, where* $\tau = \psi \circ (\mathsf{id} \times \eta)$*.*

Proof. Straightforward. □

It is possible, but more difficult, to derive such a morphism ψ from a strength τ. The problem is that there are two plausible ways to define a suitable ψ:

$$\psi_1 = {\tau_{A,B}}^* \circ C\beta_{B,CA} \circ \tau_{B,CA} \circ \beta_{CA,CB}$$

and

$$\psi_2 = C\beta_{B,A} \circ {\tau_{B,A}}^* \circ C\beta_{A,CB} \circ \tau_{A,CB}$$

These two morphisms ψ_1 and ψ_2 are not necessarily the same as they involve different orders of composition. For some monads the ordering is irrelevant, but for others it is not. As we do not wish to artificially restrict our semantics by imposing such an order, especially as both forms satisfy the requirements of Theorem 2.4.5, when we need some form of strength we will use ψ rather than τ.

The lifting monad is strong: ψ is the function that takes the *smashed product* of two domains, i.e., $\psi(\bot, d_2) = \psi(d_1, \bot) = \bot$, but $\psi(\mathsf{up}(d_1), \mathsf{up}(d_2)) = \mathsf{up}(d_1, d_2)$. The powerset functor is also strong, with $\psi(X, Y) = \{(x, y) \mid x \in X, y \in Y\}$.

2.4.5 The lifting monad in enriched categories

The lifting monad will be needed to define an adequate extensional semantics in such a way that the semantics can be easily modified to an adequate intensional semantics. That the study of non-termination and time complexity are related is no surprise; one way to determine if a part of an expression is not evaluated is to replace it with an expression that is known not to terminate. If the expression as a whole still terminates then that sub-expression must not have been evaluated.

The lifting monad, however, is only defined on **PDom** (or other closely related categories). We need a version that is not dependent on the objects being ordered sets and the morphisms being functions. One possible definition comes from knowing that LA is similar to a coproduct $A + \mathbf{1}$ with extra ordering requirements; to be precise, if U is the forgetful functor from **PDom** to **Set**, then $U(LA)$ is isomorphic (in **Set**) to $\mathbf{1} + UA$. The ordering requirements mean that the definition is limited to enriched categories, but the definition given below requires no additional properties of a category.

Definition 2.4.3 In a **PDom**-enriched category $\mathcal{C}$, the *lift* of an object A consists of an object, written as LA, together with two morphisms $\mathsf{up}_A : A \to LA$ and $\perp_A : \mathbf{1} \to LA$ such that the following hold:

1. LA is strict and $\perp_A$ is the least element of $\mathrm{Hom}(\mathbf{1}, LA)$.
2. For any morphisms: $f, g : C \to A$, $\mathsf{up}_A \circ f \leq \mathsf{up}_A \circ g$ if and only if $f \leq g$.
3. If $x : \mathbf{1} \to LA$, then either $x = \perp_A$ or, for some $x' : \mathbf{1} \to A$, $x = \mathsf{up}_A \circ x'$.
4. For any $x : \mathbf{1} \to A$, $\mathsf{up}_A \circ x \neq \perp_A$
5. For any morphisms $f : A \to B$ and $x : \mathbf{1} \to B$ with $x \circ !_A \leq f$, there exists a unique morphism $f_x : LA \to B$ such that $f_x \circ \mathsf{up}_A = f$ and $f_x \circ \perp_A = x$.

A category *has lifts* if each of its objects has a lift.

The first two requirements specify the ordering of LA: $\perp$ is the least element and all other elements of A keep the same relative order. Note that the if part of requirement #2 always holds by the monotonicity of composition. The next two enforce the separate nature of $\perp$; for most set-like categories, they will follow from the final requirement. The final requirement is a close variant of the coproduct definition. The major change is the requirement that $x \circ ! \leq f$; without it f_x may not be monotone. For functions it is equivalent to requiring that the value to which $\perp$ is mapped less than or equal to any other element in the range of f.

Lifting as defined on **PDom** satisfies the requirements for lifts just defined. Furthermore the requirements are sufficient to ensure that $(L, (-)^{\perp}, \mathsf{up})$ is a Kleisli triple, where $f^{\perp} = f_{\perp}$. Thus if we let $Lf = (\mathsf{up} \circ f)^{\perp} = \mathsf{up} \circ f_{\perp}$, L becomes a functor, and $(L, \mathsf{up}, \mathsf{down})$ is a monad, where $\mathsf{down} = \mathsf{id}^{\perp} = \mathsf{id}_{\perp}$. Also $\perp$ is a natural transformation from $\mathbf{1}$ to L.

Even if a category has lifts and products, however, L may not be strong. To get strength, we need to define an equivalent to the smashed product function.

Definition 2.4.4 A category with lifts and products has *smashed products* if for every pair of objects A and B there exists a morphism $\mathsf{smash}_{A,B}$ from $LA \times LB$ to $L(A \times B)$ such that the following properties hold (also see Figure 2.13):

1. $L\pi_1 \circ \mathsf{smash}_{A,B} \circ (\mathsf{id} \times \mathsf{up}_B) = \pi_1$

2. $\mathsf{smash}_{A,B} \circ (\mathsf{up}_A \times \mathsf{up}_B) = \mathsf{up}_{A\times B}$

3. $\mathsf{smash}_{A,B\times C} \circ (\mathsf{id}_{LA} \times \mathsf{smash}_{B,C}) \circ \alpha^r_{LA,LB,LC} = L\alpha^r_{A,B,C} \circ \mathsf{smash}_{A\times B,C} \circ (\mathsf{smash}_{A,B} \times \mathsf{id}_{LC})$

4. $\mathsf{smash}_{A',B'} \circ (f^\perp \times g^\perp) = (\mathsf{smash}_{A',B'} \circ (f \times g))^\perp \circ \mathsf{smash}_{A,B}$

5. For all objects A, B, $\mathsf{smash}_{A,B} \circ (\perp_A \times \mathsf{id}_{LB}) = \perp_{A\times B} \circ !_{1\times LB}$,

6. For all objects A, B, $\mathsf{smash}_{A,B} \circ \beta = L\beta \circ \mathsf{smash}_{B,A}$.

$L(A \times B)$ is called the *smashed product* of LA and LB.

The first four requirements are the same as four of the five requirements of strength; the last strength requirement is derivable from the rest and from the last requirement of smashed products, which show that the smashed products are order-independent. Thus in any category with lifts and smashed products the lifting monad is also strong. The remaining requirement covers the behavior of smash with $\perp$; the property that $\mathsf{smash} \circ (\mathsf{id} \times \perp)$ is also $\perp$ can be derived from the rest.

PDom has smashed products, where $\mathsf{smash}(\perp, y) = \mathsf{smash}(x, \perp) = \perp$, and

$$\mathsf{smash}(\mathsf{up}(x), \mathsf{up}(y)) = \mathsf{up}(\langle x, y\rangle)$$

Note that if we let $A' = LA$ and $B' = LB$ in **PDom** (i.e, if A' and B' are both domains), then there is a standard *smashed product* of A' and B', written $A' \otimes B'$, where

$$A' \otimes B' = \{\perp\} \cup \{\langle a, b\rangle \mid a \in A', b \in B', a \neq \perp, b \neq \perp\}$$

i.e., the smashed product identifies all pairs of the form $\langle \perp, b\rangle$ or $\langle a, \perp\rangle$ with $\perp$. It is clear that $A' \otimes B' = L(A \times B)$; therefore, the traditional smashed product for domains is an example of smashed products as defined in this section.

2.4.6 The categorical denotational semantics

Computer scientists have used category theory for denotational semantics in a variety of ways. In particular, the notion of Cartesian closure has been shown to closely model the simply-typed λ-calculus ([19]). The semantics we derive in this section also has roots in the semantics of the λ-calculus but we have made three significant changes: First, the semantics also explicitly includes additional language constructs such as recursion and conditionals. Second, our semantics not only handles additional constants, but is set up so that the constants can remain as a parameter in our framework, to be instantiated as needed. Third, the semantics models non-termination through the use of the lifting monad.

To derive the semantic function, we first start with the semantics defined in Figure 2.8 which uses domains. We then add semantic definitions for the data types not yet covered and define environments in a manner consistent with category theory. We then convert one semantic function to the other.

When possible, we convert the call-by-value and call-by-need versions in parallel. In some cases, such as with conditionals, the conversions are the same for both; for others, such as with abstractions, the conversions are quite different.

$$
\begin{tikzcd}
 & A \times B \arrow[dl, "\mathsf{up} \times \mathsf{up}"'] \arrow[dr, "\mathsf{up}"] & \\
LA \times LB \arrow[rr, "\mathsf{smash}"'] & & L(A \times B)
\end{tikzcd}
$$

$$
\begin{tikzcd}
LA \times LB \arrow[r, "\mathsf{smash}"] & L(A \times B) \arrow[d, "L\pi_1"] \\
LA \times B \arrow[u, "\mathsf{id} \times \mathsf{up}"] \arrow[r, "\pi_1"'] & LA
\end{tikzcd}
\qquad
\begin{tikzcd}
LA \times LB \arrow[r, "f^\perp \times g^\perp"] \arrow[d, "\mathsf{smash}"'] & LA' \times LB' \arrow[d, "\mathsf{smash}"] \\
L(A \times B) \arrow[r, "(\mathsf{smash} \circ (f \times g))^\perp"'] & L(A' \times B')
\end{tikzcd}
$$

$$
\begin{tikzcd}
(LA \times LB) \times LC \arrow[r, "\mathsf{smash} \times \mathsf{id}"] \arrow[d, "\alpha^r"'] & L(A \times B) \times LC \arrow[r, "\mathsf{smash}"] & L((A \times B) \times C) \arrow[d, "L\alpha^r"] \\
LA \times (LB \times LC) \arrow[r, "\mathsf{id} \times \mathsf{smash}"'] & LA \times L(B \times C) \arrow[r, "\mathsf{smash}"'] & L(A \times (B \times C))
\end{tikzcd}
$$

$$
\begin{tikzcd}
\mathbf{1} \times LB \arrow[r, "!"] \arrow[d, "\perp \times \mathsf{id}"'] & \mathbf{1} \arrow[d, "\perp"] \\
LA \times LB \arrow[r, "\mathsf{smash}"'] & L(A \times B)
\end{tikzcd}
\qquad
\begin{tikzcd}
LA \times LB \arrow[r, "\mathsf{smash}"] \arrow[d, "\beta"'] & L(A \times B) \arrow[d, "L\beta"] \\
LB \times LA \arrow[r, "\mathsf{smash}"'] & L(B \times A)
\end{tikzcd}
$$

Figure 2.13: Smashed product requirements

Types

The first notable difference concerning the meaning of types is that the denotational model assumes that all the types are strict while the categorical model adds strictness explicitly through the functor L. To have proper access to the unlifted objects, the meaning of a type τ in the categorical model ($\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$ or $\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau]\!]$) must therefore refer to the unlifted object. When we want the strict object, we use $L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$ or $L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau]\!]$. Because the denotational meanings of a type ($\mathcal{T}^{\mathrm{V}}_{\mathrm{D}}[\![\tau]\!]$ and $\mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\tau]\!]$) are always strict, when converting the semantics we compare $\mathcal{T}^{\mathrm{V}}_{\mathrm{D}}[\![\tau]\!]$ to $L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$, not to $\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$. Similarly, for the call-by-name version, $\mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\tau]\!]$ will be compared to $L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau]\!]$.

This difference requires that the meanings of ground types must also change. Instead of flat domains such as $\mathbf{N}_\perp$ and $\mathbf{B}_\perp$, we set the meaning of ground types to objects such as the discretely ordered sets $\mathbf{N}$ and $\mathbf{Bool}$. Formally, for each ground type g we let A_g be some object in $\mathcal{C}$. There are no ordering (or any other) requirements on A_g although in practice A_g will be discretely ordered except when built by non-strict constructors (in the call-by-name semantics). In section 2.4.7 we will define the meaning of **bool** for the general categorical model.

The domain model did not give meanings to constructed data types. There are, however, standard definitions for data types such as products and sums and we can use those to help determine the more general case for the categorical model. A typical definition for products defines the meaning of $\tau_1 \times \tau_2$, $\mathcal{T}^{\mathrm{V}}_{\mathrm{D}}[\![\tau_1 \times \tau_2]\!]$, to be $\mathcal{T}^{\mathrm{V}}_{\mathrm{D}}[\![\tau_1]\!] \otimes \mathcal{T}^{\mathrm{V}}_{\mathrm{D}}[\![\tau_2]\!]$ for the call-by-value semantics and $(\mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\tau_1]\!] \times \mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\tau_2]\!])_\perp$ for the call-by-name semantics, where $A \otimes B$ is the *smashed product* as defined in the previous section. The use of smashed products corresponds to the strictness of the pairing operator; if either component fails to terminate, the pair itself fails to terminate as well. For standard call-by-name semantics, pairing is non-strict, so the subparts can be $\perp$. The external lift distinguishes non-terminating pairs from a terminating pair with non-terminating components. Similarly, for sums, $\mathcal{T}^{\mathrm{V}}_{\mathrm{D}}[\![\tau_1 + \tau_2]\!]$ is the smashed sum $\mathcal{T}^{\mathrm{V}}_{\mathrm{D}}[\![\tau_1]\!] \oplus \mathcal{T}^{\mathrm{V}}_{\mathrm{D}}[\![\tau_2]\!]$ while $\mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\tau_1 + \tau_2]\!]$ is the separated sum $\mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\tau_1]\!] + \mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\tau_2]\!]$.

With lifts, it is simple to represent smashed products, smashed sums, and separated sums categorically. The smashed product of LA and LB is $L(A \times B)$, while the smashed sum is $L(A+B)$. Similarly, the separated sum of LA and LB is $L(LA + LB)$. Therefore we can directly translate the meanings just given to the categorical setting:

$$\begin{array}{ll}
\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1 \times \tau_2]\!] = \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!] \times \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_2]\!] & \quad \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 \times \tau_2]\!] = L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!] \times L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!] \\
\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1 + \tau_2]\!] = \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!] + \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_2]\!] & \quad \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 + \tau_2]\!] = L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!] + L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]
\end{array}$$

The converted version above suggests a good choice for the general case: Suppose that for each constructed data type function δ of n arguments there exists some n-ary functor F_δ on $\mathcal{C}$. For products $F_\times(A, B)$ would be $A \times B$, and for sums $F_+(A, B)$ would be $A + B$. Then $\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\delta(\tau_1, \ldots, \tau_n)]\!]$ would be $F_\delta(\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!], \ldots, \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_n]\!])$ and $\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\delta(\tau_1, \ldots, \tau_n)]\!]$ would be $F_\delta(L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!], \ldots, L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_n]\!])$. Conceptually, as all call-by-value data types are evaluated strictly, we know that the data types will always be constructed from terminating arguments; we therefore do not need to include the possibility that an argument might not terminate. Conversely, for the call-by-name data types, the arguments may not be evaluated when a data type is constructed; we must therefore include the possibility of non-termination when interpreting the data type.

Lastly, we must determine the meaning of functional types. For the call-by-name semantics the translation is straightforward: As $\mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\tau_1 \to \tau_2]\!] = [\mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\tau_1]\!] \Rightarrow \mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\tau_2]\!]]_\perp$ given that $\mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\tau]\!]$ is translated to $L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau]\!]$, then we should specify that

$$L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 \to \tau_2]\!] = L[L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!] \Rightarrow L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]] \quad \text{and} \quad \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 \to \tau_2]\!] = [L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!] \Rightarrow L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]]$$

For the call-by-value semantics the translation is more complicated because $\mathcal{T}_D^V[\![\tau_1 \to \tau_2]\!]$ is the domain $[\mathcal{T}_D^V[\![\tau_1]\!] \Rightarrow^\circ \mathcal{T}_D^V[\![\tau_2]\!]]$. The problem is that the domain of strict continuous functions does not directly correspond to categorical exponentiation. We can, however, derive a meaning for call-by-value functional types without resorting to the call-by-name meaning (which would anyway later cause problems when figuring out the meaning of abstractions). Because we always know a strict function's value on $\perp$, a strict function is uniquely defined by its action on the unlifted portion of its input. This means that $[A \Rightarrow LB]$ and $[LA \Rightarrow^\circ LB]$ are isomorphic in **Dom**, with isomorphisms $\phi : [LA \Rightarrow^\circ LB] \to [A \Rightarrow LB]$, defined as $\phi(f)(a) = f(\mathsf{up}(a))$, and $\phi^{-1} : [A \Rightarrow LB] \to [LA \Rightarrow^\circ LB]$, defined as $\phi^{-1}(g) = g^\perp$. Now we have access to a categorical form of exponentiation, so we can set

$$\mathcal{T}_E^V[\![\tau_1 \to \tau_2]\!] = [\mathcal{T}_E^V[\![\tau_1]\!] \Rightarrow L\mathcal{T}_E^V[\![\tau_2]\!]]$$

Conceptually, since the arguments to call-by-value functions are always evaluated first, the only part of a function we need to model is its behavior on terminating inputs; we already know its behavior on non-terminating inputs.

Environments

For domains an environment ρ is a usually defined as partial function from variables to a domain consisting of the meaning of all types. The standard categorical definition of an environment is typically an n-ary product; given a type environment $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, an environment object consistent with Γ is the product $\mathcal{T}_D^V[\![\tau_1]\!] \times \ldots \times \mathcal{T}_D^V[\![\tau_n]\!]$ (or, more precisely, $\times(\mathcal{T}_D^V[\![\tau_1]\!], \ldots, \mathcal{T}_D^V[\![\tau_n]\!])$). For a given Γ the two definitions have an obvious correspondence in **Dom**; applying an environment to x_i corresponds to taking the i'th projection.

Therefore for the call-by-value semantics, let

$$\mathcal{T}_E^V[\![\Gamma]\!] = \times(L\mathcal{T}_E^V[\![\tau_1]\!], \ldots, L\mathcal{T}_E^V[\![\tau_n]\!])$$

The call-by-name semantics treats environments identically, so let

$$\mathcal{T}_E^N[\![\Gamma]\!] = \times(L\mathcal{T}_E^N[\![\tau_1]\!], \ldots, L\mathcal{T}_E^N[\![\tau_n]\!])$$

There is some question as to whether we need to include L in the call-by-value semantics, as variables derived from abstractions are guaranteed to be bound to terminating values. Variables derived from recursion, however, have no such guarantee, and so we must include the possibility that variables do not terminate in some environments. This is because we are allowing the general form of recursion, `rec` $z.e$, instead of a common but more restricted form where e must be a syntactic value of functional type.

With these definitions of types and environment, given an expression e such that $\Gamma \vdash e : \tau$, its call-by-value meaning, $\mathcal{V}_E[\![\Gamma \vdash e : \tau]\!]$, is a morphism from $\mathcal{T}_E^V[\![\Gamma]\!]$ to $L\mathcal{T}_E^V[\![\tau]\!]$. Similarly its call-by-name meaning, $\mathcal{N}_E[\![\Gamma \vdash e : \tau]\!]$, is a morphism from $\mathcal{T}_E^N[\![\Gamma]\!]$ to $L\mathcal{T}_E^N[\![\tau]\!]$. The next several sections describe how we derive the categorical meaning of an expression from the domain-theoretic meaning.

Variables

There is nothing particularly unusual in the definition of variables for either semantics; as noted, if $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, the equivalent to applying an environment ρ to a variable x_i is taking the i'th projection. Therefore let

$$\mathcal{V}_E[\![\Gamma \vdash x_i : \tau_i]\!] = \pi_i^n : \mathcal{T}_E^V[\![\Gamma]\!] \to L\mathcal{T}_E^V[\![\tau_i]\!]$$
$$\mathcal{N}_E[\![\Gamma \vdash x_i : \tau_i]\!] = \pi_i^n : \mathcal{T}_E^N[\![\Gamma]\!] \to L\mathcal{T}_E^N[\![\tau_i]\!]$$

2.4.7 Abstraction

To convert the meaning of abstractions we need to make several changes: we need to convert to the product form of environments, we need to convert to a categorical form of exponentiation (currying as opposed to function definition), and, for the call-by-value semantics, we need to convert to the altered form of the exponentiation itself.

We first convert the call-by-name semantics, as its conversion is simpler. From Figure 2.8, the domain-theoretic definition was given as

$$\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash \mathtt{lam}\ x.e : \tau' \to \tau]\!]\rho = \mathsf{up}(\lambda d.\mathcal{N}_{\mathrm{D}}[\![\Gamma, x : \tau' \vdash e : \tau]\!]\rho[x \mapsto d])$$

For product environments, the result of appending a value d to an environment can be represented as a pair $\langle \rho, d \rangle$, giving us

$$\mathsf{up}(\lambda d.\mathcal{N}_{\mathrm{D}}[\![\Gamma, x : \tau' \vdash e : \tau]\!]\langle \rho, d \rangle)$$

Furthermore, in **Dom**, $\mathsf{curry}(f)(x) = \lambda y.f\langle x, y \rangle$, so by shifting to curry we get

$$\mathsf{up}(\mathsf{curry}(\mathcal{N}_{\mathrm{D}}[\![\Gamma, x : \tau' \vdash e : \tau]\!])\rho)$$

and, by noting that, in **Dom**, $(f \circ g)(x) = f(g(x))$, we end up with

$$\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash \mathtt{lam}\ x.e : \tau' \to \tau]\!]\rho = (\mathsf{up} \circ \mathsf{curry}(\mathcal{N}_{\mathrm{D}}[\![\Gamma, x : \tau' \vdash e : \tau]\!]))\rho$$

We can now safely drop ρ from both sides of the equation. Thus we let

$$\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{lam}\ x.e : \tau' \to \tau]\!] = \mathsf{up} \circ \mathsf{curry}(\mathcal{N}_{\mathrm{E}}[\![\Gamma, x : \tau' \vdash e : \tau]\!])$$

For call-by-value, the meaning of an abstraction given in Figure 2.8 was

$$\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash \mathtt{lam}\ x.e : \tau' \to \tau]\!]\rho = \mathsf{up}(\mathsf{strict}(\lambda d.\mathcal{V}_{\mathrm{D}}[\![\Gamma, x : \tau' \vdash e : \tau]\!]\rho[x \mapsto d]))$$

Using the same conversions just used for the call-by-name case we get

$$\mathsf{up} \circ (\mathsf{strict}(\mathsf{curry}(\mathcal{V}_{\mathrm{D}}[\![\Gamma, x : \tau' \vdash e : \tau]\!])\rho))$$

To get a result showing the desired form it is necessary to both remove the strict function and move the environment ρ to a place where we can drop it safely. First we note that the function $\mathsf{strict}(\mathsf{curry}(\mathcal{V}_{\mathrm{D}}[\![\Gamma, x : \tau' \vdash e : \tau']\!])\rho)$ is an element of $[\mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau']\!] \Rightarrow^{\circ} \mathcal{T}_{\mathrm{D}}^{\mathrm{V}}[\![\tau]\!]]$ and what we want is an element of $[\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau']\!] \Rightarrow L\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]]$. Let f be a morphism from $A \times LB$ to LC. Then applying the isomorphism ϕ (between $[LB \Rightarrow^{\circ} LC]$ and $[B \Rightarrow LC]$) to $\mathsf{strict}(\mathsf{curry}(f)\rho)$, we get that for any $b \in B$,

$$\begin{aligned}
\phi(\mathsf{strict}(\mathsf{curry}(f)\rho))b = {} & \mathsf{strict}(\mathsf{curry}(f)\rho)(\mathsf{up}(b)) \\
= {} & \mathsf{curry}(f)\rho(\mathsf{up}(b)) \\
= {} & f(\rho, \mathsf{up}(b)) \\
= {} & (f \circ (\mathsf{id} \times \mathsf{up}))(\rho, b) \\
= {} & \mathsf{curry}(f \circ (\mathsf{id} \times \mathsf{up}))\rho b
\end{aligned}$$

Therefore $\phi(\mathsf{strict}(\mathsf{curry}(f)\rho)) = \mathsf{curry}(f \circ (\mathsf{id} \times \mathsf{up}))\rho$. Now ρ can safely be dropped from both sides of the equation leaving

$$\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{lam}\ x.e : \tau' \to \tau]\!] = \mathsf{up} \circ \mathsf{curry}(\mathcal{V}_{\mathrm{E}}[\![\Gamma, x : \tau' \vdash e : \tau]\!] \circ (\mathsf{id} \times \mathsf{up}))$$

Application

The primary complication involved in converting the meaning of an application comes from the use of down in both the call-by-name and call-by-value semantics. In the categorical setting, while we can define a morphism down from $L(LA)$ to LA for any object A, there is no guarantee that there is such a morphism from LA to A even if A is strict. Therefore we cannot directly convert down. Instead we get the same effect by using the lifting operation $(-)^{\perp}$ on the result. To see this, let $f : A \to L[LB \Rightarrow LC]$ in **Dom** and let $g : A \to LB$. Then for any $a \in A$

$$\begin{aligned}
(\mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle f, \mathsf{up} \circ g\rangle)(a) &= \mathsf{app}^{\perp}(\mathsf{smash}(f(a), \mathsf{up}(g(a)))) \\
&= \begin{cases} \mathsf{app}^{\perp}(\perp) & \text{if } f(a) = \perp \\ \mathsf{app}^{\perp}(\mathsf{up}(f', g(a))) & \text{if } f(a) = \mathsf{up}(f') \end{cases} \\
&= \begin{cases} \perp & \text{if } f(a) = \perp \\ \mathsf{app}(f', g(a)) & \text{if } f(a) = \mathsf{up}(f') \end{cases} \\
&= (\mathsf{down}(fa))(ga)
\end{aligned}$$

Therefore we can substitute $\mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle f, \mathsf{up} \circ g\rangle$, which is well defined for any Cartesian closed category with lifts and smashed products, for $\mathsf{app} \circ \langle \mathsf{down} \circ f, g\rangle$. For the call-by-name semantics, this substitution is all we need, because Figure 2.8 defined application as

$$\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_1(e_2) : \tau]\!]\rho = \quad (\mathsf{down}(\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!]\rho))(\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_2 : \tau']\!]\rho)$$

We therefore can define call-by-name application as

$$\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1(e_2) : \tau]\!] = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!], \mathsf{up} \circ \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau']\!]\rangle$$

The meaning given for call-by-value application,

$$\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_1(e_2) : \tau]\!]\rho = (\mathsf{down}(\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!]\rho))(\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_2 : \tau']\!]\rho)$$

has the same form, but $\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!]\rho$ is an element of $L[\mathcal{T}^{\mathrm{V}}_{\mathrm{D}}[\![\tau']\!] \Rightarrow^{\circ} \mathcal{T}^{\mathrm{V}}_{\mathrm{D}}[\![\tau]\!]]$ so we also need to convert it to an element of a domain of the form $L[A \Rightarrow LB]$. We can manage the conversion with the isomorphism ϕ^{-1}, obtaining

$$\mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle L\phi^{-1} \circ \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!], \mathsf{up} \circ \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau']\!]\rangle$$

but ϕ^{-1} is only defined for domains. We still need a categorical form that is equivalent (for domains) to the last equation. By case analysis we can show that, for domains,

$$\mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle L\phi^{-1} \circ f, \mathsf{up} \circ g\rangle = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle f, g\rangle$$

which lacks ϕ^{-1}. To see this, let $f : A \to L[B \Rightarrow LC]$, $g : A \to LB$ and let $a \in A$. Then if $f(a) = \mathsf{up}(f')$ and $g(a) = \mathsf{up}(g')$,

$$\begin{aligned}
&(\mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle L\phi^{-1} \circ f, \mathsf{up} \circ g\rangle)(a) \\
&\quad = \mathsf{app}^{\perp}(\mathsf{smash}(L\phi^{-1}(\mathsf{up}(f')), \mathsf{up}(\mathsf{up}(g')))) \\
&\quad = \mathsf{app}^{\perp}(\mathsf{smash}(\mathsf{up}(f'^{\perp}), \mathsf{up}(\mathsf{up}(g')))) \\
&\quad = \mathsf{app}(f'^{\perp}, \mathsf{up}(g')) \\
&\quad = \mathsf{app}(f', g') \\
&\quad = (\mathsf{app}^{\perp} \circ \mathsf{smash})(\mathsf{up}(f'), \mathsf{up}(g')) \\
&\quad = (\mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle f, g\rangle)(a)
\end{aligned}$$

Otherwise, if $f(a) = \mathsf{up}(f')$ and $g(a) = \bot$, then

$$\begin{array}{rl}(\mathsf{app}^\bot \circ \mathsf{smash} \circ \langle L\phi^{-1} \circ f, \mathsf{up} \circ g\rangle)(a) & \\ = & \mathsf{app}^\bot(\mathsf{smash}(\mathsf{up}((f')^\bot), \mathsf{up}(\bot))) \\ = & \mathsf{app}((f')^\bot, \bot) \\ = & \bot \\ = & (\mathsf{app}^\bot \circ \mathsf{smash})(\mathsf{up}(f'), \bot) \\ = & (\mathsf{app}^\bot \circ \mathsf{smash} \circ \langle f, g\rangle)(a)\end{array}$$

Lastly, if $f(a) = \bot$, then

$$\begin{array}{rl}(\mathsf{app}^\bot \circ \mathsf{smash} \circ \langle L\phi^{-1} \circ f, \mathsf{up} \circ g\rangle)(a) & \\ = & \mathsf{app}^\bot(\mathsf{smash}(L\phi^{-1}(\bot), \mathsf{up}(g(a)))) \\ = & \bot \\ = & (\mathsf{app}^\bot \circ \mathsf{smash} \circ \langle f, g\rangle)(a)\end{array}$$

We have thus explored all possible cases for $f(a)$ and $g(a)$, so we have shown that there is an equivalent form for call-by-value application that is not dependent on domains, namely

$$\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1(e_2) : \tau]\!] = \mathsf{app}^\bot \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau']\!]\rangle$$

Conditionals

The call-by-value meaning of a conditional expression is

$$\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau]\!]\rho = \begin{cases} \mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_2 : \tau]\!]\rho & \text{if } \mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!]\rho = \mathrm{tt} \\ \mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_3 : \tau]\!]\rho & \text{if } \mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!]\rho = \mathrm{ff} \\ \bot & \text{if } \mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!]\rho = \bot \end{cases}$$

and the call-by-name meaning is similar. To convert this to a categorical setting we need to define a boolean object in the category and separate the third case above out so that we can use the lifting structure instead. Both $\mathcal{T}^{\mathrm{V}}_{\mathrm{D}}[\![\mathbf{bool}]\!]$ and $\mathcal{T}^{\mathrm{N}}_{\mathrm{D}}[\![\mathbf{bool}]\!]$ are the flat domain $\{\mathsf{tt}, \mathsf{ff}, \bot\}$, with $\bot$ the least element. Therefore we would expect that $\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\mathbf{bool}]\!]$ and $\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\mathbf{bool}]\!]$ would be something equivalent to the discretely ordered domain $\{\mathsf{tt}, \mathsf{ff}\}$. We need, however, a categorical definition of booleans, so assume that $\mathbf{B}$ is some object in the category with two morphisms, $\mathsf{tt}, \mathsf{ff} : \mathbf{1} \to \mathbf{B}$, and assume that for each object A, there exists a morphism cond_A from $\mathbf{B} \times (A \times A)$ to A such that

- $\mathsf{cond}_A \circ \langle \mathsf{tt} \circ !_{A\times A}, \mathsf{id}_{A\times A}\rangle = \pi_1$, and
- $\mathsf{cond}_A \circ \langle \mathsf{ff} \circ !_{A\times A}, \mathsf{id}_{A\times A}\rangle = \pi_2$

As for the ordering, assume that tt and ff are unrelated in the predomain $\mathrm{Hom}(\mathbf{1}, \mathbf{B})$. Then we can say that $\mathbf{B}$ is a boolean object. Note that there may be several valid boolean objects for any category.

Any Cartesian closed category that has coproducts for which $\mathsf{inl} : \mathbf{1} \to \mathbf{1}+\mathbf{1}$ and $\mathsf{inr} : \mathbf{1} \to \mathbf{1}+\mathbf{1}$ are unrelated has a boolean object, namely $\mathbf{B} = \mathbf{1} + \mathbf{1}$, with $\mathsf{tt} = \mathsf{inl}$, $\mathsf{ff} = \mathsf{inr}$, and

$$\mathsf{cond}_A = \mathsf{app} \circ ([\mathsf{curry}(\pi_1 \circ \pi_2), \mathsf{curry}(\pi_2 \circ \pi_2)] \times \mathsf{id}_{A\times A})$$

In **PDom**, $\mathbf{1}+\mathbf{1}$ is isomorphic to the discrete domain $\{\mathsf{tt}, \mathsf{ff}\}$; and the definition of cond satisfying the requirements is

$$\mathsf{cond}(x, \langle a, b\rangle) = \begin{cases} a & \text{if } x = \mathsf{tt} \\ b & \text{if } x = \mathsf{ff} \end{cases}$$

The next step is to convert the standard definition to one using cond. As with application, the case where the input is $\perp$ can be handled by lifting cond. The function $\mathsf{cond}^{\perp}$, however, has as input an object of the form $L(\mathbf{B} \times (LA \times LA))$ (A must be lifted in order for $\mathsf{cond}^{\perp}$ to be well-defined). From the meanings of the subparts of the conditional expression it is possible to construct a morphism from the environment object to an object of the form $L\mathbf{B} \times (LA \times LA)$. Then we can use the morphism $\mathsf{smash} \circ (\mathsf{id} \times \mathsf{up})$ to get from $L\mathbf{B} \times (LA \times LA)$ to $L(\mathbf{B} \times (LA \times LA))$. Therefore a tentative categorical meaning for conditionals is (for call-by-value)

$$\begin{aligned} &\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau]\!] \\ &\quad = \mathsf{cond}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!], \mathsf{up} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_3 : \tau]\!]\rangle\rangle \end{aligned}$$

We will now show that the above equation is equivalent to the original definition.

Let $z \in L\mathbf{B}$ and $a_1, a_2 \in LA$. Then

$$\begin{aligned} &\mathsf{cond}^{\perp} \circ \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up})(z, \langle a_1, a_2\rangle) \\ &\quad = \mathsf{cond}^{\perp}(\mathsf{smash}(z, \mathsf{up}(\langle a_1, a_2\rangle))) \\ &\quad = \begin{cases} \mathsf{cond}^{\perp}(\perp) & \text{if } z = \perp \\ \mathsf{cond}^{\perp}(\mathsf{up}(b, \langle a_1, a_2\rangle)) & \text{if } z = \mathsf{up}(b) \end{cases} \\ &\quad = \begin{cases} \perp & \text{if } z = \perp \\ \mathsf{cond}(b, \langle a_1, a_2\rangle) & \text{if } z = \mathsf{up}(b) \end{cases} \\ &\quad = \begin{cases} \perp & \text{if } z = \perp \\ a_1 & \text{if } z = \mathsf{up}(\mathsf{tt}) \\ a_2 & \text{if } z = \mathsf{up}(\mathsf{ff}) \end{cases} \end{aligned}$$

We can derive the meaning of conditionals for call-by-name in a similar manner, obtaining

$$\begin{aligned} &\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau]\!] \\ &\quad = \mathsf{cond}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!], \mathsf{up} \circ \langle \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!], \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_3 : \tau]\!]\rangle\rangle \end{aligned}$$

The verification that his formulation is equivalent to the non-categorical definition is straightforward and omitted.

Recursion

The definition given for a recursive expression `rec` $x.e$ is the same for both call-by-name and call-by-value; if for each environment ρ, $f(\rho)$ is the meaning of e, the meaning of `rec` $x.e$ is

$$\mathsf{fix}(\lambda d. f(\rho[x \mapsto d]))$$

Using products for environments and the morphism fix defined in Chapter 2 this becomes

$$\mathsf{fix}(\lambda d. f\langle \rho, d\rangle)$$

or

$$\mathsf{fix} \circ \mathsf{curry}(f)$$

We can also write the above construction using $\mathsf{fixp}_{D,D'} : \mathrm{Hom}(D' \times D, D) \to \mathrm{Hom}(D', D)$, leading to a simpler but equivalent definition. Thus let

$$\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \Gamma : \mathtt{rec}\ x.e]\!]\tau = \mathsf{fixp}(\mathcal{V}_{\mathrm{E}}[\![\Gamma, x : \tau \vdash e : \tau]\!])$$

and

$$\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash \Gamma : \mathtt{rec}\ x.e]\!]\tau = \mathsf{fixp}(\mathcal{N}_{\mathrm{E}}[\![\Gamma, x : \tau \vdash e : \tau]\!])$$

Constants

One standard method for defining the meaning of arity 0 constants (for either the call-by-name or call-by-value) is to have a separate function $\mathcal{C}[\![-]\!]$ that would take a constant c of type τ to a morphism from $\mathbf{1}$ to $\mathcal{T}[\![\tau]\!]$. The meaning of c would then be $\mathcal{C}[\![c : \tau]\!] \circ !$. The original meanings of `true` and `false`, tt and ff respectively, have such a form. With the different interpretation of booleans, their meanings translate to $\mathsf{up} \circ \mathsf{tt} \circ !$ and $\mathsf{up} \circ \mathsf{ff} \circ !$.

For constants with an arity greater than 0 in most cases a semantic definition is given only for the constant applied to all its arguments. For example, the semantic definitions in [11] for the product constants are

$$\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash <e_1, e_2> : \tau_1 \times \tau_2]\!]\rho = \mathsf{smash}(\mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \tau_1]\!]\rho, \mathcal{V}_{\mathrm{D}}[\![\Gamma \vdash e_2 : \tau_2]\!]\rho)$$

for the call-by-value semantics, and

$$\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash <e_1, e_2> : \tau_1 \times \tau_2]\!]\rho = \mathsf{up}(\mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_1 : \tau_1]\!]\rho, \mathcal{N}_{\mathrm{D}}[\![\Gamma \vdash e_2 : \tau_2]\!]\rho)$$

Because the language used in this dissertation includes partially applied constants we needed a method for handling them as well. It is, of course, possible to use just the standard definition of arity 0 constants including functional meanings for functional constants. There are two problems with this approach. The first is that the meanings become quite complicated; for example, `fst`, of type $\tau_1 \times \tau_2 \to \tau_1$, would have as its meaning $\mathsf{up} \circ \mathsf{curry}(\mathsf{up} \circ \pi_1 \circ \pi_2)$. For pairing, the meaning would be even more complicated. The other problem is that the action of a constant function is not dependent on the particular constant itself until all arguments are present and evaluated, so determining properties such as soundness for each constant would entail much redundancy. Thus it is simpler to define the meaning of a constant as a morphism from the product of all its argument objects to the resulting object; that is, if c has arity n and type $\tau_1 \to \ldots \to \tau_n \to \tau$, then the call-by-name meaning of c, $\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![c : \tau]\!]$, would be a morphism from $\times(L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!], \ldots, L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_n]\!])$ to $L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau]\!]$. For the call-by-value meaning, we know the behavior of c applied to any non-terminating expression so we do not need to lift the meaning of the input types. Thus $\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c : \tau]\!]$ would be a morphism from $\times(\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!], \ldots, \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_n]\!])$ to $L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$.

We still need to convert the meaning of a constant into a valid meaning of an expression consisting of a constant. To get the general meaning of a constant in the call-by-value semantics, we need to convert $\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c : \tau]\!]$ to a morphism from $\mathbf{1}$ to $L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$. For a constant of arity 1 and type $\tau' \to \tau$, this requires converting a morphism from $\times(\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau']\!])$ (i.e., $\mathbf{1} \times \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau']\!]$) to $L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$ to a morphism from $\mathbf{1}$ to $L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau' \to \tau]\!]$ (i.e., $\mathbf{1}$ to $L[\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau']\!] \Rightarrow L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]]$). Currying gives us a morphism from $\mathbf{1}$ to $[\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau']\!] \Rightarrow L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]]$; We can then use up to convert to $L[\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau']\!] \Rightarrow L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]]$. The use of up also corresponds to the operational property that a solitary constant always successfully evaluates (to itself).

Now suppose that c is a constant of arity 2 and type $\tau_1 \to \tau_2 \to \tau$. Then $\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c : \tau]\!]$ is a morphism from $\times(\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!], \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_2]\!])$ to $L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$, where

$$\times(\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!], \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_2]\!]) = (\mathbf{1} \times \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!]) \times \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_2]\!]$$

We need to convert this to a morphism from $\mathbf{1}$ to $L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1 \to \tau_2 \to \tau_3]\!]$, which is equal to $L[\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!] \Rightarrow L[\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_2]\!] \Rightarrow L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]]]$. It is clear that $\mathsf{up} \circ \mathsf{curry}(\mathsf{up} \circ \mathsf{curry}(\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c : \tau]\!]))$ is such a morphism.

Therefore we can convert $\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c : \tau]\!]$ by repeatedly applying the function $\mathsf{raise}_\perp(f) = \mathsf{up} \circ \mathsf{curry}(f)$. To see that $\mathsf{raise}_\perp$ has the proper form, first let $[A \Rightarrow^L B]$ denote $[A \Rightarrow LB]$. Then we can write $\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1 \to \ldots \to \tau_n]\!]$ more succinctly as $[\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!] \Rightarrow^L \ldots \Rightarrow^L \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_n]\!]]$.

Theorem 2.4.6 *If $f : (\ldots(A_0 \times A_1) \times \ldots) \times A_n \to LA$, then*

$$\mathsf{raise}^{(n)}_\perp(f) : A_0 \to L[A_1 \Rightarrow^L \ldots \Rightarrow^L A_n \Rightarrow^L A]$$

Proof. By induction on n. If $n = 0$, then $\mathsf{raise}^{(0)}_\perp(f) = f : A_0 \to LA$ as desired. Now assume that the lemma holds for $n - 1 \geq 0$. Then, by the induction hypothesis,

$$\mathsf{raise}^{(n-1)}_\perp(f) : A_0 \times A_1 \to L[A_2 \Rightarrow^L \ldots \Rightarrow^L A_n \Rightarrow^L A]$$

so

$$\mathsf{curry}(\mathsf{raise}^{(n-1)}_\perp(f)) : A_0 \to [A_1 \Rightarrow L[A_2 \Rightarrow^L \ldots \Rightarrow^L A_n \Rightarrow^L A]]$$

i.e.,

$$\mathsf{curry}(\mathsf{raise}^{(n-1)}_\perp(f)) : A_0 \to [A_1 \Rightarrow^L \ldots \Rightarrow^L A_n \Rightarrow^L A]$$

To get a morphism from A_0 to $L[A_1 \Rightarrow^L \ldots \Rightarrow^L A_n \Rightarrow^L A]$ we can just compose with η. Therefore $\mathsf{raise}^{(n)}_\perp(f) = \eta \circ \mathsf{curry}(\mathsf{raise}^{(n-1)}_\perp(f))$ has the desired form. $\square$

As a special case, the above lemma shows that if $f : \times(\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!], \ldots \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_n]\!]) \to L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$, then $\mathsf{raise}^{(n)}_\perp(f)$ is a morphism from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1 \to \tau_n \to \tau]\!]$. Therefore a possible meaning for constants is

$$\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash c : \tau]\!] = \mathsf{raise}^{(\mathrm{ar}(c))}_\perp(\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c : \tau]\!]) \circ !$$

With this definition $\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{true} : \mathbf{bool}]\!]$ would be $\mathsf{up} \circ \mathsf{tt}$, so $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{true} : \mathbf{bool}]\!]$ would be $\mathsf{up} \circ \mathsf{tt} \circ !$ as expected.

To see that the above definition is reasonable for constants of higher arity, let us look again at the meaning of pairing. Since `pair` has arity 2, $\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{pair} : \tau_1 \times \tau_2]\!]$ must be a morphism from $(\mathbf{1} \times \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!]) \times \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_2]\!]$ to $L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1 \times \tau_2]\!] = L(\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!] \times \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_2]\!])$. An obvious value to give for $\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{pair}]\!]$ is the isomorphism $\mathsf{up} \circ (\pi_2 \times \mathsf{id})$. We must now show that this definition is equivalent to the definition taken from [11], i.e., if $\rho : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\Gamma]\!]$, then $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{pair}\ e_1 e_2 : \tau_1 \times \tau_2]\!] \circ \rho$ should be $\mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau_1]\!] \circ \rho, \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau_2]\!] \circ \rho \rangle$. We verify this as follows:

First, if $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau_2]\!] \circ \rho$ is $\perp$, then

$$\begin{aligned}
&\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{pair}\ e_1 e_2 : \tau_1 \times \tau_2]\!] \circ \rho \\
&\quad = \mathsf{app}^\perp \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{pair}\ e_1 : \tau_2 \to \tau_1 \times \tau_2]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau_2]\!] \rangle \circ \rho \\
&\quad = \mathsf{app}^\perp \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{pair}\ e_1 : \tau_2 \to \tau_1 \times \tau_2]\!] \circ \rho, \perp \rangle \\
&\quad = \mathsf{app}^\perp \circ \perp \\
&\quad = \perp
\end{aligned}$$

Similarly, if $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau_1]\!] \circ \rho = \bot$, then

$$\begin{aligned}
&\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{pair}\ e_1 e_2 : \tau_1 \times \tau_2]\!] \circ \rho \\
&\quad = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{pair}\ e_1 : \tau_2 \to \tau_1 \times \tau_2]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau_2]\!]\rangle \circ \rho \\
&\quad = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{pair} : \tau_1 \to \tau_2 \to \tau_1 \times \tau_2]\!], \\
&\qquad\quad \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau_1]\!]\rangle \circ \rho, \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau_2]\!] \circ \rho\rangle \\
&\quad = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{pair} : \tau_1 \to \tau_2 \to \tau_1 \times \tau_2]\!], \bot\rangle, \\
&\qquad\quad \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau_2]\!] \circ \rho\rangle \\
&\quad = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \bot, \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau_2]\!] \circ \rho\rangle \\
&\quad = \bot
\end{aligned}$$

Note that the above equations hold regardless of the constant's value; they reflect the call-by-value nature of the semantics.

Lastly suppose that for some $y_1 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!]$ and $y_2 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_2]\!]$, $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau_1]\!] \circ \rho = \mathsf{up} \circ y_1$ and $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau_2]\!] \circ \rho = \mathsf{up} \circ y_2$. Then

$$\begin{aligned}
&\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{pair}\ e_1 e_2 : \tau_1 \times \tau_2]\!] \circ \rho \\
&\quad = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{pair} : \tau_1 \to \tau_2 \to \tau_1 \times \tau_2]\!] \circ \rho, \\
&\qquad\quad \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau_1]\!] \circ \rho\rangle, \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau_2]\!] \circ \rho\rangle \\
&\quad = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{raise}^{(2)}_{\perp}(\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{pair} : \tau_1 \to \tau_2 \to \tau_1 \times \tau_2]\!]) \circ ! \circ \rho, \\
&\qquad\quad \mathsf{up} \circ y_1\rangle, \mathsf{up} \circ y_2\rangle \\
&\quad = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{up} \circ \mathsf{curry}(\mathsf{raise}^{(1)}_{\perp}(\mathsf{up} \circ (\pi_2 \times \mathsf{id}))) \circ !, \mathsf{up} \circ y_1\rangle, \mathsf{up} \circ y_2\rangle \\
&\quad = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{app}^{\perp} \circ \mathsf{up} \circ \langle \mathsf{curry}(\mathsf{raise}^{(1)}_{\perp}(\mathsf{up} \circ (\pi_2 \times \mathsf{id}))) \circ !, y_1\rangle, \mathsf{up} \circ y_2\rangle \\
&\quad = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{app} \circ \langle \mathsf{curry}(\mathsf{raise}^{(1)}_{\perp}(\mathsf{up} \circ (\pi_2 \times \mathsf{id}))) \circ !, y_1\rangle, \mathsf{up} \circ y_2\rangle \\
&\quad = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{raise}^{(1)}_{\perp}(\mathsf{up} \circ (\pi_2 \times \mathsf{id})) \circ \langle !, y_1\rangle, \mathsf{up} \circ y_2\rangle \\
&\quad = \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{up} \circ \mathsf{curry}(\mathsf{up} \circ (\pi_2 \times \mathsf{id})) \circ \langle !, y_1\rangle, \mathsf{up} \circ y_2\rangle \\
&\quad = \mathsf{up} \circ (\pi_2 \times \mathsf{id}) \circ \langle\langle !, y_1\rangle, y_1\rangle \\
&\quad = \mathsf{up} \circ \langle y_1, y_2\rangle
\end{aligned}$$

which is the smashed product.

We still need to convert the call-by-name meaning of a constant, $\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![c]\!]$, to a morphism from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{N}}[\![\tau_1 \to \ldots \to \tau_n \to \tau]\!]$. The function $\mathsf{raise}_{\perp}$ also successfully performs the conversion; if c has type $\tau' \to \tau$, then $\mathsf{raise}_{\perp}(\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![c]\!])$ is a morphism from $\mathbf{1}$ to $L[L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau']\!] \Rightarrow L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau]\!]] = L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau' \to \tau]\!]$. Essentially the additional L found in an argument to a higher-arity constant is matched by the additional L found in the input to an element of functional type.

Figure 2.14 contains the categorical semantics for both the call-by-name and call-by-value semantics.

2.5 Soundness

A denotational semantics is sound relative to an operational semantics if the meanings given to expressions are consistent with the rules for the operational semantics. For example if an operational semantics declares that two expressions are equivalent (e.g., `if true then` e_1 `else` $e_2 \equiv e_1$) then we would expect that the denotational semantics gives both expressions the same value. For our particular operational semantics, we interpret operational properties such as "e evaluates to v" as specifying that the (final) value of e is v, so we would expect that the denotational semantics gives the same meaning to e as to v. Formally,

$$
\begin{array}{rcl}
\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![g]\!] & = & A_g \\
\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\delta(\tau_1,\ldots,\tau_n)]\!] & = & F_\delta(\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!],\ldots,\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_n]\!]) \\
\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1 \to \tau_2]\!] & = & [\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!] \Rightarrow L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_2]\!]] \\
\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![x_1:\tau_1,\ldots,x_n:\tau_n]\!] & = & \times(L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!],\ldots,L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_n]\!])
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash c:\tau]\!] & = & \mathsf{raise}^{(\mathrm{ar}(c))}_{\perp}(\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c:\tau]\!]) \circ !_{\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\Gamma]\!]} \\
\mathcal{V}_{\mathrm{E}}[\![x_1:\tau_1,\ldots,x_n:\tau_n \vdash x_i:\tau_i]\!] & = & \pi^n_i \\
\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{lam}\ x.e:\tau' \to \tau]\!] & = & \mathsf{up} \circ \mathsf{curry}(\mathcal{V}_{\mathrm{E}}[\![\Gamma, x:\tau' \vdash e:\tau]\!] \circ (\mathsf{id} \times \mathsf{up})) \\
\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1(e_2):\tau]\!] & = & \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1:\tau' \to \tau]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2:\tau']\!]\rangle \\
\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3:\tau]\!] & = & \mathsf{cond}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1:\mathbf{bool}]\!], \\
& & \qquad \mathsf{up} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2:\tau]\!], \\
& & \qquad\qquad \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_3:\tau]\!]\rangle\rangle \\
\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{rec}\ x.e:\tau]\!] & = & \mathsf{fixp}(\mathcal{V}_{\mathrm{E}}[\![\Gamma, x:\tau \vdash e:\tau]\!]) \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{true}:\mathbf{bool}]\!] & = & \mathsf{up} \circ \mathsf{tt} \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{false}:\mathbf{bool}]\!] & = & \mathsf{up} \circ \mathsf{ff}
\end{array}
$$

The categorical call-by-value semantics

$$
\begin{array}{rcl}
\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![g]\!] & = & A_g \\
\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\delta(\tau_1,\ldots,\tau_n)]\!] & = & F_\delta(L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!],\ldots,L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_n]\!]) \\
\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 \to \tau_2]\!] & = & [L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!] \Rightarrow L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]] \\
\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![x_1:\tau_1,\ldots,x_n:\tau_n]\!] & = & \times(L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!],\ldots,L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_n]\!])
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash c:\tau]\!] & = & \mathsf{raise}^{(\mathrm{ar}(c))}_{\perp}(\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![c:\tau]\!]) \circ !_{\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\Gamma]\!]} \\
\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash x_i:\tau_i]\!] & = & \pi^n_i \\
\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{lam}\ x.e:\tau' \to \tau]\!] & = & \mathsf{up} \circ \mathsf{curry}(\mathcal{N}_{\mathrm{E}}[\![\Gamma, x:\tau' \vdash e:\tau]\!]) \\
\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1(e_2):\tau]\!] & = & \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1:\tau' \to \tau]\!], \\
& & \qquad \mathsf{up} \circ \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_2:\tau']\!]\rangle \\
\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3:\tau]\!] & = & \mathsf{cond}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1:\mathbf{bool}]\!], \\
& & \qquad \mathsf{up} \circ \langle \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_2:\tau]\!], \\
& & \qquad\qquad \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_3:\tau]\!]\rangle\rangle \\
\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{rec}\ x.e:\tau]\!] & = & \mathsf{fixp}(\mathcal{N}_{\mathrm{E}}[\![\Gamma, x:\tau \vdash e:\tau]\!]) \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{true}:\mathbf{bool}]\!] & = & \mathsf{up} \circ \mathsf{tt} \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{false}:\mathbf{bool}]\!] & = & \mathsf{up} \circ \mathsf{ff}
\end{array}
$$

The categorical call-by-name semantics

Figure 2.14: Denotational semantics using category theory and the lifting monad

Definition 2.5.1 For the call-by-value evaluation strategy, a denotational semantics V is *sound* if for all closed expressions e of type τ, $e \Rightarrow_{\mathrm{v}} v$ implies that

$$\mathcal{V}[\![\vdash e : \tau]\!] = \mathcal{V}_{\mathrm{E}}[\![\vdash v : \tau]\!]$$

For the call-by-name evaluation strategy, a denotational semantics N is sound if for all closed expressions e of type τ, $e \Rightarrow_{\mathrm{n}} v$ implies that

$$\mathcal{N}[\![\vdash e : \tau]\!] = \mathcal{N}_{\mathrm{E}}[\![\vdash v : \tau]\!]$$

We use the term "extensional" to apply to the semantics found in this chapter because the semantics does not distinguish between an expression and its final value. With an *intensional* semantics we are also interested in properties of an expression besides its final value. Therefore we would expect that even when $e \Rightarrow_{\mathrm{v}} v$ the intensional meanings of e and v may differ. The difference, however, should be directly related to the intensional properties described by the intensional operational semantics.

Given an intensional and an extensional semantics for the same language, the intensional semantics is *separable* if the extensional semantics can be extracted from it. We will show in the next few chapters that separability also implies that the extensional semantics is sound whenever the intensional semantics is sound. Intuitively, intensional soundness compares both the internal and external properties, so if we remove the intensional part of the semantics we should still expect that the extensional parts maintain the desired relationship. Therefore we will postpone the proofs that $\mathcal{V}_{\mathrm{E}}$ is sound relative to the call-by-value operation semantics ant that $\mathcal{N}_{\mathrm{E}}$ is sound relative to the call-by-name extensional semantics until we have first shown that the intensional versions are sound.

2.6 Adequacy

A denotational semantics is *adequate* relative to an operational semantics if expressions that fail to terminate in the operational semantics are precisely the expressions that denote $\perp$ in the denotational semantics, that is, the denotation meaning of an expression e is not $\perp$ if and only the expression terminates in the operation semantics. Because we based our categorical semantics on an adequate domain-based semantics, we would expect our categorical semantics to be adequate as well. It is, and the proof that it is adequate is similar to the proof that the domain-based semantics is adequate. Therefore we will only summarize the proofs in this section; the full proofs can be found in Appendix A. Unlike soundness, we do need a formal proof of the adequacy of the extensional semantics because we derive the adequacy of the intensional semantics from the adequacy of the extensional semantics.

The proof that the domain-based semantics is adequate comes from [11]. To alter it into a proof that our semantics is adequate requires changes in two main areas: the difference between a domain-theoretic formulation and our category-theoretic formulation and the use of unspecified constants and data structures. The change from domain theory to category theory involves only minor details; the addition of unspecified constants and data structures, however, requires some new assumptions. The assumptions made about constants differ slightly between the proofs for the call-by-name and call-by-value semantics, but the assumptions made about data types are independent of the evaluation strategy.

For data types, we need to be able to examine subparts of both the syntactic expressions and their categorical meanings. We do this by defining deconstructors. Let δ be a n-ary type constructor and F_δ the related n-ary functor in $\mathcal{C}$. Then assume that for each $1 \leq i \leq n$, there is a set P_i of natural transformations such that for each i, each $p \in P_i$, and any objects $A_1, \ldots, A_n$, $p_{A_1,\ldots,A_n}$ is a morphism from $F_\delta(A_1, \ldots, A_n)$ to LA_i. These transformations allow us to recover the subparts of a data type. The requirement of naturality guarantees that the deconstructors depend only on the structure of the datatype and not its contents. For example, in the product functor $F_\times(A, B) = A \times B$, there are two deconstructors: $p_1 = \mathsf{up}_A \circ \pi_1$ and $p_2 = \mathsf{up}_B \circ \pi_2$. We lift the result of a deconstructor so that we can handle cases where a datatype may not always "contain" a substructure of a certain type. For example, if $\mathcal{C}$ has coproducts and $F_+(A, B)$ is $A + B$, then $\mathsf{inl} \circ a$ does not contain any elements from B and $\mathsf{inr} \circ b$ does not contain any elements from A. Thus to define deconstructors to both A and B, we must consider the possibly that there is no relevant value of A or B in the original element. Such cases are represented by $\perp$ in LA or LB. Therefore, the two deconstructors for $F_+(A, B)$ are $p_1 = [\mathsf{up}_A, \perp \circ !_B]$ and $p_2 = [\perp \circ !_A, \mathsf{up}_B]$. We include potentially multiple deconstructors for each i because for some data types, such as arrays or lists, there may be more than one component for each subtype.

Each denotational deconstructor has an equivalent syntactic deconstructor used to perform the same action but using expressions not meanings. Therefore assume that for each deconstructor p there exists an expression e_p with one free variable x such that $x : \delta(\tau_1, \ldots, \tau_n) \vdash e_p : \tau_i$. For example, given the definitions of p_1 and p_2 in the previous paragraph let $e_{p_1} = \mathtt{fst}(x)$ and $e_{p_2} = \mathtt{snd}(x)$. At the general level we do not need to explicitly specify how the value of e_p relates to p because the definition of adequacy for constructed data types implicitly includes the relationship between the syntactic and categorical deconstructors. In practice, however, we find that the following relation holds given a deconstructor p from type $\delta(\tau_1, \ldots, \tau_n)$ to τ_i, its related expression e_p, an a closed expression e of type $\delta(\tau_1, \ldots, \tau_n)$:

$$p^\perp \circ \mathcal{V}_{\mathrm{E}}[\![e : \delta(\tau_1, \ldots, \tau_n)]\!] = \mathcal{V}_{\mathrm{E}}[\![[e/x]e_p : \tau_i]\!]$$

for the call-by-value semantics, and

$$(\mathsf{down} \circ p)^\perp \circ \mathcal{N}_{\mathrm{E}}[\![e : \delta(\tau_1, \ldots, \tau_n)]\!] = \mathcal{N}_{\mathrm{E}}[\![[e/x]e_p : \tau_i]\!]$$

for the call-by-name semantics. For the first product deconstructor p_1 this is the same as saying that

$$L\pi_1 \circ \mathcal{V}_{\mathrm{E}}[\![e : \tau_1 \times \tau_2]\!] = \mathcal{V}_{\mathrm{E}}[\![\mathtt{fst}(e) : \tau_1]\!]$$

for the call-by-value semantics, and

$${\pi_1}^\perp \circ \mathcal{N}_{\mathrm{E}}[\![e : \tau_1 \times \tau_2]\!] = \mathcal{N}_{\mathrm{E}}[\![\mathtt{fst}(e) : \tau_1]\!]$$

for the call-by-name semantics, i.e., that taking the first projection via a deconstructor is equivalent to applying `fst`.

In addition to examining subparts, we need to be able to describe how types are constructed from the subparts, in case there is more than one possible construction. This corresponds to types where we must do a case analysis to determine its structure, as in coproducts or lists, instead of just pulling out it subparts, as in products. The adequacy proof contains a relation $\precsim^{\mathrm{V}}$ between expressions and their meanings and the relation must not hold unless expressions of a given data type are combined in a way compatible with their meanings. We want, however, only to examine which of several possible constructions were actually used, and to ignore the actual contents.

A simple method for describing structure without content is to use the object $F_\delta(\mathbf{1}, \ldots, \mathbf{1})$. The object $F_\delta(\mathbf{1}, \ldots, \mathbf{1})$ contains the changeable structural information of F_δ without containing any information particular to subparts themselves. Then given any element $z : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\delta(\tau_1, \ldots, \tau_n)]\!]$, its structure can be described by composing z with $LF_\delta(!, \ldots, !)$. In particular, to compare z with a closed value v of type $\delta(\tau_1, \ldots, \tau_n)$, we simply compare the morphisms $LF_\delta(!, \ldots, !) \circ z$ and $LF_\delta(!, \ldots, !) \circ \mathcal{V}_{\mathrm{E}}[\![v : \delta(\tau_1, \ldots, \tau_n)]\!]$. For products, $F_\times(\mathbf{1}, \mathbf{1}) = \mathbf{1} \times \mathbf{1} \cong \mathbf{1}$, indicating that there is no need to check amongst various possible constructions. For lists, however, $F_{\mathbf{list}}(\mathbf{1}, \mathbf{1})$ is isomorphic to the discrete set of integers, indicating that when comparing lists we need to compare length as well as contents.

2.6.1 Call-by-value adequacy

The general plan of the proof is to define a type-indexed family of relations $\precsim_\tau^{\mathrm{V}}$ between morphisms z from $\mathbf{1}$ to $L\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ and closed expressions e of type τ such that $z \precsim_\tau^{\mathrm{V}} e$ and $z \neq \bot$ implies that $e \Rightarrow_{\mathrm{v}} v$ for some value v. Then we show that this relation holds between expressions and their meanings. To define this relationship, we define a related family of relationships, $\precsim_\tau^{\mathrm{V}*}$ between morphisms from $\mathbf{1}$ to $\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ and values of type τ. These relationships are thus define by mutual induction on the structure of the type τ:

Definition 2.6.1 For each type τ, let $\precsim_\tau^{\mathrm{V}}$ be a relation between morphisms from $\mathbf{1}$ to $L\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ and closed expressions of type τ, defined as follows: given $z : \mathbf{1} \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ and $\vdash e : \tau$, $z \precsim_\tau^{\mathrm{V}} e$ if

1. $z = \bot_{\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]}$, or

2. There exists a closed value v of type τ and a morphism $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ such that $e \Rightarrow_{\mathrm{v}} v$, $z = \mathsf{up} \circ y$, and $y \precsim_\tau^{\mathrm{V}*} v$, where $\precsim_\tau^{\mathrm{V}*}$ is a relation between morphisms from $\mathbf{1}$ to $\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ and closed values of type τ, defined as follows:

   - $y \precsim_g^{\mathrm{V}*} v$ if $y \leq \mathcal{V}_{\mathrm{E}}[\![v : \tau]\!]$
   - $y \precsim_{\delta(\tau_1,\ldots,\tau_n)}^{\mathrm{V}*} v$ if for each deconstructor $p \in P_i$, $p \circ y \precsim_{\tau_i}^{\mathrm{V}} [v/x]e_p$ and if
   $$\mathsf{up} \circ F_\delta(!, \ldots, !) \circ z \leq LF_\delta(!, \ldots, !) \circ \mathcal{V}_{\mathrm{E}}[\![v : \delta(\tau_1, \ldots, \tau_n)]\!]$$
   - $y \precsim_{\tau' \to \tau}^{\mathrm{V}*} v$ if for all $y' : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau']\!]$ and closed values v' such that $y' \precsim_{\tau'}^{\mathrm{V}*} v'$,
   $$\mathsf{app} \circ \langle y, y' \rangle \precsim_\tau^{\mathrm{V}} v(v')$$

Because $\precsim_\tau^{\mathrm{V}}$ is defined in terms of $\precsim_\tau^{\mathrm{V}*}$, and $\precsim_\tau^{\mathrm{V}*}$ is defined in terms of $\precsim_{\tau'}^{\mathrm{V}}$, where τ' is a structural subpart of τ, the relationships are well defined.

It is straightforward to show that if v is a closed value of type τ, and $z \precsim_\tau^{\mathrm{V}} v$, then either $z = \bot$ or for some $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$, $z = \mathsf{up} \circ y$ and $y \precsim_\tau^{\mathrm{V}*} v$. Similarly, if v is a closed value of type τ, then $y \precsim_\tau^{\mathrm{V}*} v$ implies that $\mathsf{up} \circ y \precsim_\tau^{\mathrm{V}} v$.

In section 2.7.5 we show how the rules for type constructors translate to an more intuitive definition for coproducts, and in Appendix A we show the same for products and lists.

The adequacy proof follows from the proof that $\mathcal{V}_{\mathrm{E}}[\![e : \tau]\!] \precsim_\tau^{\mathrm{V}} e$ for all (well typed) closed expressions. That proof depends on some assumptions made about constants. We could simply state that $\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![c]\!]$ is adequate, but then proving that individual constants are adequate would require redundant proofs that partially applied constants are adequate. Instead we make assumptions only about the fully applied constants and prove that the assumptions imply that partially applied constants are adequate as well.

Definition 2.6.2 For any constant c of type $\tau_1 \to \ldots \to \tau_n \to \tau$ and arity n, a morphism f from $\times(\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1]\!], \ldots, \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_n]\!])$ to $L\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ is *adequate for* c if, for all morphisms $y_i : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_i]\!]$ $(1 \leq i \leq n)$ and closed values v_i of type τ_i such that $\mathtt{up} \circ y_i \precsim_{\tau_i}^{\mathrm{V}} v_i$,

$$f \circ \langle\rangle(y_1, \ldots, y_n) \precsim_{\tau}^{\mathrm{V}} cv_1 \ldots v_n$$

By this definition, $\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{true} : \mathbf{bool}]\!]$ is adequate for `true` and $\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{false} : \mathbf{bool}]\!]$ is adequate for `false`.

The rest of the proof that $\mathcal{V}_{\mathrm{E}}[\![e]\!] \precsim_{\tau}^{\mathrm{V}} e$ is shown in Appendix A. With this lemma, we can now prove adequacy:

Theorem 2.6.1 (Adequacy, part 1) *Suppose that* $\vdash e : \tau$. *Then* $\mathcal{V}_{\mathrm{E}}[\![e]\!] \neq \bot$ *implies that there exists a value* v *such that* $e \Rightarrow_{\mathrm{v}} v$.

Proof. Suppose that $\mathcal{V}_{\mathrm{E}}[\![e]\!] \neq \bot$. Then as $\mathcal{V}_{\mathrm{E}}[\![e]\!] \precsim_{\tau}^{\mathrm{V}} e$, there exists a value v such that $e \Rightarrow_{\mathrm{v}} v$. □

Adequacy also requires that if $e \Rightarrow_{\mathrm{v}} v$ then $\mathcal{V}_{\mathrm{E}}[\![e]\!] \neq \bot$. To prove this we need soundness and the property that the meaning of a syntactic values is never $\bot$. As we have not yet formally proven soundness, we must delay this portion of the proof to the next chapter.

2.6.2 Call-by-name adequacy

The proof of adequacy for the call-by-name semantics is very similar to the proof for the call-by-value semantics. One primary difference is that we cannot necessarily describe the structure of a value v of type $\delta(\tau_1, \ldots, \tau_n)$ as $LF_\delta(!, \ldots, !) \circ \mathcal{N}_{\mathrm{E}}[\![v : \delta(\tau_1, \ldots, \tau_n)]\!]$. The reason is that the subparts of v may be unevaluated expressions, and, if v is also recursively defined, we may need to know adequacy of the call-by-name semantics in order to prove adequacy, leading to a circular proof.

To avoid this problem we define, for each constructed type $\delta(\tau_1, \ldots, \tau_n)$, a function $\mathcal{K}_\delta[\![-]\!]$ from values to morphisms from $\mathbf{1}$ to $F_\delta(\mathbf{1}, \ldots, \mathbf{1})$. We can then compare the structure of a morphism z from $\mathbf{1}$ to $L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\delta(\tau_1, \ldots, \tau_n)]\!]$ and a value $v : \delta(\tau_1, \ldots, \tau_n)$ by comparing the morphisms $L(F_\delta(!, \ldots, !)) \circ z$ and $\mathtt{up} \circ \mathcal{K}_\delta[\![v : \delta(\tau_1, \ldots, \tau_n)]\!]$. Our original structure definition,

$$LF_\delta(!, \ldots, !) \circ \mathcal{N}_{\mathrm{E}}[\![v : \delta(\tau_1, \ldots, \tau_n)]\!]$$

is then equivalent to $\mathtt{up} \circ \mathcal{K}_\delta[\![v : \delta(\tau_1, \ldots, \tau_n)]\!]$. For example, for any value v of type $\tau_1 \times \tau_2$, $\mathcal{K}_\times[\![v : \tau_1 \times \tau_2]\!] : \mathbf{1} \to \mathbf{1} \times \mathbf{1}$ is $\langle \mathbf{1}, \mathbf{1} \rangle$ (which, in the case of products, is the only possible morphism as $\mathbf{1} \times \mathbf{1}$ is a terminal object). See section B.2.4 for the (recursive) definition used for lists.

Appendix B contains the details of the call-by-name adequacy proof.

2.7 The FL constants

When we defined the extensional semantics using category theory we made certain assumptions about the meanings of ground types, constructed data types, and their constants. In this section we will show that we can satisfy these assumptions for all the types and constants used in FL. Because we are proving that assumptions are satisfied, in this section (and whenever we are explicitly dealing with FL) we will be explicitly working with the category **PDom**. Thus we can use prior knowledge of domains to aid in proving the assumptions.

There were no particular assumptions made about ground types themselves (except that relevant objects exist) but for constructed data types we need to show that the construction is a functor. As we have deferred the proof of soundness to future chapters we have not yet made any assumptions relating to soundness. We will, however, have to show that the constants are adequate; we will do so for a few constants at the end of this section and defer the rest to Appendices A and B.

2.7.1 Natural numbers

The denotational meaning of **nat** ($A_{\mathbf{nat}}$) is the discretely ordered predomain of natural numbers $\mathbf{N} = \{0, 1, \ldots\}$. For each $m \in \mathbf{N}$, let $\mathsf{n}_m : \mathbf{1} \to \mathbf{N}$ represent the element m. Let plus be the morphism from $\mathbf{N}\times\mathbf{N}$ to $\mathbf{N}$, defined by the function $\mathsf{plus}(n, m) = n{+}m$ (i.e., $\mathsf{plus}\circ\langle \mathsf{n}_n, \mathsf{n}_m\rangle = \mathsf{n}_{n+m}$). Similarly, $\mathsf{times}(n, m) = n * m$, and $\mathsf{minus}(n, m) = 0$ if $n < m$ and $n - m$ otherwise. Lastly let $\mathsf{leq}(n, m) = \mathsf{tt}$ if $n \leq m$ and ff otherwise, and let $\mathsf{eq}(n, m) = \mathsf{tt}$ if $n = m$ and ff otherwise. Because $\mathbf{N}$ (and thus $\mathbf{N}\times\mathbf{N}$) is discretely ordered, all functions from $\mathbf{N}$ or $\mathbf{N}\times\mathbf{N}$ are automatically continuous and thus are valid morphisms in **PDom**.

Figure 2.15 contains the definitions of the constants given for natural numbers: first the call-by-value meanings followed by the call-by-name meanings. The product (iso-)morphisms seen in these definitions are used to convert the general product $\times()$ to forms needed by the specific constants. Interestingly, if we define an n-ary version of smash, where

$$\mathsf{nsmash}(0) = \mathsf{up} \quad \text{and} \quad \mathsf{nsmash}(n) = \mathsf{smash} \circ (\mathsf{nsmash}(n-1) \times \mathsf{id})$$

then for all the constants c in Figure 2.15,

$$\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![c]\!] = \mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c]\!]^{\perp} \circ \mathsf{nsmash}(n)$$

where n is the arity of c. This relationship holds for all ground type constants which evaluate their arguments strictly.

2.7.2 Products

For products, we take advantage of the knowledge that **PDom** has categorical products. Therefore we simply set $F_\times$ to be the product bifunctor. Then we can easily derive the meaning of the product constants from the product constructions and morphisms, as seen in Figure 2.16. The extra complexity in the call-by-name meaning of `pair` comes from treating pairing strictly while not treating the meaning of the product type strictly; if `pair` were a lazy constant its meaning would be simply $\mathsf{up} \circ (\pi_2 \times \mathsf{id})$, the same as its call-by-value meaning (although with different domains and co-domains).

2.7.3 Sums

Just as we can represent products in FL with categorical products in **PDom**, we can represent sums in FL with categorical coproducts. Therefore let F_+ be the coproduct functor $-+-$. The standard coproduct constructs, however, are not designed to handle combinations of products and coproducts easily. We can, however, use them to define an intermediate function on morphisms. What we need is a general construction to handle case analysis on objects of the form $(A + A') \times B$ as well as the usual construction $(A + A')$. We can define such a construction for any Cartesian closed category. Specifically, for morphisms $f : A \times B \to C$ and $f' : A' \times B \to C$, let $\mathsf{case}(f, f') : (A + A') \times B \to C$ be $\mathsf{uncurry}([\mathsf{curry}(f), \mathsf{curry}(f')])$. As a function in **PDom**, this definition becomes

$$\mathsf{case}(f, f')(x, b) = \begin{cases} f(a, b) & \text{if } x = \mathsf{inl}(a) \\ f'(a', b) & \text{if } x = \mathsf{inr}(a') \end{cases}$$

The relevant properties of case are that $\mathsf{case}(f, f') \circ (\mathsf{inl} \times \mathsf{id}) = f$ and $\mathsf{case}(f, f') \circ (\mathsf{inr} \times \mathsf{id}) = f'$.

With case we now can easily define the meaning for all the sum constants, listed in Figure 2.17. Because `inl` and `inr` are lazy constructors, their meanings are the same for the call-by-value and call-by-name semantics. The call-by-name meaning of `case` is quite different largely because we want to avoid evaluating the second and third arguments unless we know they will be used.

$$
\begin{aligned}
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\overline{n}]\!] &= \mathsf{up} \circ \mathsf{n}_n \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![=]\!] &= \mathsf{up} \circ \mathsf{eq} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\leq]\!] &= \mathsf{up} \circ \mathsf{leq} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![+]\!] &= \mathsf{up} \circ \mathsf{plus} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\dot{-}]\!] &= \mathsf{up} \circ \mathsf{minus} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\times]\!] &= \mathsf{up} \circ \mathsf{times} \circ (\pi_2 \times \mathsf{id})
\end{aligned}
$$

Call by value

$$
\begin{aligned}
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\overline{n}]\!] &= \mathsf{up} \circ \mathsf{n}_n \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![=]\!] &= L\mathsf{eq} \circ \mathsf{smash} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\leq]\!] &= L\mathsf{leq} \circ \mathsf{smash} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![+]\!] &= L\mathsf{plus} \circ \mathsf{smash} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\dot{-}]\!] &= L\mathsf{minus} \circ \mathsf{smash} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\times]\!] &= L\mathsf{times} \circ \mathsf{smash} \circ \circ(\pi_2 \times \mathsf{id})
\end{aligned}
$$

Call-by-name

Figure 2.15: Extensional meanings of the integer constants

$$
\begin{aligned}
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{pair}]\!] &= \mathsf{up} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{fst}]\!] &= \mathsf{up} \circ \pi_1 \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{snd}]\!] &= \mathsf{up} \circ \pi_2 \circ \pi_2
\end{aligned}
$$

Call-by-value

$$
\begin{aligned}
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{pair}]\!] &= L(\mathsf{up} \times \mathsf{up}) \circ \mathsf{smash} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{fst}]\!] &= {\pi_1}^{\perp} \circ \pi_2 \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{snd}]\!] &= {\pi_2}^{\perp} \circ \pi_2
\end{aligned}
$$

Call-by-name

Figure 2.16: Extensional meanings of the product constants

$$
\begin{aligned}
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{inl}]\!] &= \mathsf{up} \circ \mathsf{inl} \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{inr}]\!] &= \mathsf{up} \circ \mathsf{inr} \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{case}]\!] &= \mathsf{case}(\mathsf{vleft}, \mathsf{vright}) \circ (\pi_2 \times \mathsf{id}) \circ \alpha^r \\
\mathsf{vleft} &= \mathsf{app} \circ \beta \circ (\mathsf{id} \times \pi_1) \\
\mathsf{vright} &= \mathsf{app} \circ \beta \circ (\mathsf{id} \times \pi_2)
\end{aligned}
$$

Call-by-value

$$
\begin{aligned}
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{inl}]\!] &= \mathsf{up} \circ \mathsf{inl} \circ \pi_2 \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{inr}]\!] &= \mathsf{up} \circ \mathsf{inr} \circ \pi_2 \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{case}]\!] &= (\mathsf{case}(\mathsf{nleft}, \mathsf{nright}))^{\perp} \circ \mathsf{smash} \circ (\pi_2 \times \mathsf{up}) \circ \alpha^r \\
\mathsf{nleft} &= \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \beta \circ (\mathsf{up} \times \pi_1) \\
\mathsf{nright} &= \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \beta \circ (\mathsf{up} \times \pi_2)
\end{aligned}
$$

Call-by-name

Figure 2.17: Extensional meanings of the sum constants

2.7.4 Lists

Unlike sums and products, we have not already defined a construction in **PDom** suitable for lists. Because it is a recursive data type, its definition is slightly more complicated. Furthermore, the meaning of the list type differs between the call-by-value and call-by-name semantics. For the purposes of defining constants, in both cases we will need a functor List on **PDom** and morphisms related to the constants `nil`, `cons`, etc. There are two general methods for defining lists in **PDom**: First, we could simply give a direct definition of a domain and define the morphisms as functions. Second, we could define the predomain of lists as the solution to a recursive domain equation. While the first method is more understandable in the short run, the latter method allows us to change the definition easily when we encounter both lazy lists (for the call-by-name) and lazy lists with costs.

Given an object A, let $\mathsf{List}\,(A)$ be a solution to the equivalence

$$\mathsf{List}\,(A) \cong \mathbf{1} + (A \times \mathsf{List}\,(A)) \tag{2.2}$$

where $\cong$ indicates that the two predomains are related by an isomorphism, and, for a morphism $f : A \to B$, let $\mathsf{List}\,(f)$ be the (least) solution to the equation

$$\mathsf{List}\,(f) = \mathsf{fold}_B \circ (\mathsf{id} + (f \times \mathsf{List}\,(f))) \circ \mathsf{unfold}_A \tag{2.3}$$

where $\mathsf{fold}_A : \mathbf{1}{+}A{\times}\mathsf{List}\,(A) \to \mathsf{List}\,(A)$ and $\mathsf{unfold}_A : \mathsf{List}\,(A) \to \mathbf{1}{+}A{\times}\mathsf{List}\,(A)$ are the isomorphism described earlier.

The theory of recursive domain equations tells us that both equations have (least) solutions. In particular, given a domain A, the set $\{[a_1, \ldots, a_n] \mid n \geq 0,\ a_1, \ldots, a_n \in A\}$, ordered element-wise, is a solution to equation 2.2. In that case

$$\mathsf{fold}_A(\mathsf{inl}(*)) = [\,], \quad \mathsf{fold}_A(\mathsf{inr}(a, [a_1, \ldots, a_n])) = [a, a_1, \ldots, a_n]$$

and unfold_A is the obvious inverse. Furthermore we can see that the mapping function,

$$\mathsf{map}(f)[a_1, \ldots, a_n] = [f(a_1), \ldots, f(a_n)]$$

is the least solution to equation 2.3.

With List, fold and unfold, we now have enough information to define the list constants. Let

$$\begin{array}{rcl} \mathsf{nil}_A &=& \mathsf{fold}_A \circ \mathsf{inl} : \mathbf{1} \to \mathsf{List}\,(A) \\ \mathsf{hd}_A &=& [\bot_A, \mathsf{up}_A \circ \pi_1] \circ \mathsf{unfold}_A : \mathsf{List}\,(A) \to LA \\ \mathsf{cons}_A &=& \mathsf{fold}_A \circ \mathsf{inr} : A \times \mathsf{List}\,(A) \to \mathsf{List}\,(A) \\ \mathsf{tl}_A &=& [\bot_A, \mathsf{up} \circ \pi_2] \circ \mathsf{unfold}_A : \mathsf{List}\,(A) \to L\mathsf{List}\,(A) \\ \mathsf{null?}_A &=& [\mathsf{tt}, \mathsf{ff} \circ !] \circ \mathsf{unfold}_A : \mathsf{List}\,(A) \to \mathbf{B} \end{array}$$

hd and tl need to lift their results because otherwise they would be undefined on empty lists.

For the call-by-name semantics, we need a form of lazy lists. In particular as `cons` does not evaluate its arguments, lists must include the possibility of having no tail (represented as having $\bot$ for a tail). We also will need to include infinite lists as they can be represented by such expressions as `rec l.true::l`. We can define such a domain as a (least) solution to the domain equation

$$\mathsf{Llist}\,(A) \cong \mathbf{1} + (A \times L\mathsf{Llist}\,(A))$$

with lfold_A and $\mathsf{lunfold}_A$ as the isomorphisms and, for $f : A \to B$, $\mathsf{Llist}\,(f)$ as the (least) solution to the equation

$$\mathsf{Llist}\,(f) = \mathsf{lfold}_B \circ (\mathsf{id}_1 + (f \times L\mathsf{Llist}\,(f))) \circ \mathsf{lunfold}_A$$

Again from domain theory we know that the equations have solutions; one such solution is to let $\mathsf{Llist}\,(A)$ be

$$\mathsf{List}\,(A) \cup \{[a_1, \ldots, a_n, \bot] \mid n > 0 \text{ and } a_1, \ldots, a_n \in A\} \cup \{[a_1, a_2, \ldots] \mid \forall n > 0, a_n \in A\}$$

The lists are ordered element-wise, except that a list l of the form $[a_1, \ldots, a_n, \bot]$ is less than any list of at least n elements whose first n elements are greater than the element in the same position of l. With the these definitions established we can define equivalent morphisms for the list constants:

$$\begin{array}{rcl} \mathsf{lnil}_A &=& \mathsf{lfold}_A \circ \mathsf{inl} : \mathbf{1} \to \mathsf{Llist}\,(A) \\ \mathsf{lhd}_A &=& [\bot_A, \mathsf{up}_A \circ \pi_1] \circ \mathsf{lunfold}_A : \mathsf{Llist}\,(A) \to LA \\ \mathsf{lcons}_A &=& \mathsf{lfold}_A \circ \mathsf{inr} : A \times L\mathsf{Llist}\,(A) \to \mathsf{Llist}\,(A) \\ \mathsf{ltl}_A &=& [\bot_A, \pi_2] \circ \mathsf{lunfold}_A : \mathsf{Llist}\,(A) \to L\mathsf{Llist}\,(A) \\ \mathsf{lnull?}_A &=& [\mathsf{tt}, \mathsf{ff} \circ !] \circ \mathsf{lunfold}_A : \mathsf{Llist}\,(A) \to \mathbf{B} \end{array}$$

These definitions are the same as the ones for the strict lists except for ltl_A, indicating that the tail of a lazy list is the primary difference between the two types of list.

In practice, we care little about the precise definition of the list object and care more about how the previous morphisms interact with each other and with $\mathsf{List}\,(f)$ or $\mathsf{Llist}\,(f)$. These interactions are listed in Figure 2.18; for most of this dissertation we will limit ourselves to these properties. Note that the morphisms hd, lhd, etc., are all natural transformations. As they affect the list without regards to the contents, they are precisely the type of morphisms we would expect to be natural.

Figure 2.19 contains the meanings for the list constants themselves. The call-by-name meaning of `head` contains a double lift because not only do we have to handle the case where the input to `head` is $\bot$, but the case where the head of the list is $\bot$ as well.

$$
\begin{array}{rclcrcl}
\mathsf{hd}_A \circ \mathsf{nil}_A & = & \bot_A & & \mathsf{lhd}_A \circ \mathsf{lnil}_A & = & \bot_A \\
\mathsf{hd}_A \circ \mathsf{cons}_A & = & \mathsf{up}_A \circ \pi_1 & & \mathsf{lhd}_A \circ \mathsf{lcons}_A & = & \mathsf{up}_A \circ \pi_1 \\
\mathsf{hd}_B \circ \mathsf{List}\, f & = & Lf \circ \mathsf{hd}_A & & \mathsf{lhd}_B \circ \mathsf{Llist}\, f & = & Lf \circ \mathsf{lhd}_A \\
\mathsf{tl}_A \circ \mathsf{nil}_A & = & \bot_A & & \mathsf{ltl}_A \circ \mathsf{lnil}_A & = & \bot_A \\
\mathsf{tl}_A \circ \mathsf{cons}_A & = & \mathsf{up}_A \circ \pi_2 & & \mathsf{ltl}_A \circ \mathsf{lcons}_A & = & \pi_2 \\
\mathsf{tl}_B \circ \mathsf{List}\, f & = & L\mathsf{List}\, f \circ \mathsf{tl}_A & & \mathsf{ltl}_B \circ \mathsf{Llist}\, f & = & L\mathsf{Llist}\, f \circ \mathsf{ltl}_A \\
\mathsf{null?}_A \circ \mathsf{nil}_A & = & \mathsf{tt} & & \mathsf{lnull?}_A \circ \mathsf{lnil}_A & = & \mathsf{tt} \\
\mathsf{null?}_A \circ \mathsf{cons}_A & = & \mathsf{ff} \circ ! & & \mathsf{lnull?}_A \circ \mathsf{lcons}_A & = & \mathsf{ff} \circ ! \\
\mathsf{null?}_B \circ \mathsf{List}\, f & = & \mathsf{null?}_A & & \mathsf{lnull?}_B \circ \mathsf{Llist}\, f & = & \mathsf{lnull?}_A \\
\mathsf{List}\, f \circ \mathsf{nil}_A & = & \mathsf{nil}_B & & \mathsf{Llist}\, f \circ \mathsf{lnil}_A & = & \mathsf{lnil}_B \\
\mathsf{List}\, f \circ \mathsf{cons}_A & = & \mathsf{cons}_B \circ (f \times \mathsf{List}\, f) & & \mathsf{Llist}\, f \circ \mathsf{lcons}_A & = & \mathsf{lcons}_B \circ (f \times L\mathsf{Llist}\, f)
\end{array}
$$

Figure 2.18: Properties of strict and lazy lists

$$
\begin{array}{rcl}
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{nil}]\!] & = & \mathsf{up} \circ \mathsf{nil} \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{cons}]\!] & = & \mathsf{up} \circ \mathsf{cons} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{head}]\!] & = & \mathsf{hd} \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{tail}]\!] & = & \mathsf{tl} \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{nil?}]\!] & = & \mathsf{up} \circ \mathsf{null?} \circ \pi_2
\end{array}
$$

Call-by-value

$$
\begin{array}{rcl}
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{nil}]\!] & = & \mathsf{up} \circ \mathsf{lnil} \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{cons}]\!] & = & \mathsf{up} \circ \mathsf{lcons} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{head}]\!] & = & (\mathsf{id}^{\bot} \circ \mathsf{lhd})^{\bot} \circ \pi_2 \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{tail}]\!] & = & \mathsf{ltl}^{\bot} \circ \pi_2 \\
\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{nil?}]\!] & = & L\mathsf{lnull?} \circ \pi_2
\end{array}
$$

Call-by-name

Figure 2.19: Extensional meanings of the list constants

2.7.5 Adequacy of the extensional constants

The proof that the extensional semantics is adequate depends both on the definition of deconstructors for the data types and on the assumptions made about constants. While we leave the details of the adequacy proof to the appendices, in this section we will discuss definitions of the deconstructors and will show how we can use them to convert the general adequacy assumptions into specific conditions that resemble familiar adequacy requirements.

For deconstructors, there are three data types affected: products, sums, and lists. Products and sums each need two deconstructors, one for each subtype. Therefore let $P_1 = \{p_{\times_1}\}$ where $p_{\times_1} : A \times B \to LA$ is $\mathsf{up}_A \circ \pi_1$ and let $P_2 = \{p_{\times_2}\}$ where $p_{\times_2} : A \times B \to LB$ is $\mathsf{up}_B \circ \pi_2$. Similarly for sums let $P_1 = \{p_{+_1}\}$ where $p_{+_1} : A + B \to LA$ is $[\mathsf{up}_A, \bot \circ !_B]$ and let $P_2 = \{p_{+_2}\}$ where $p_{+_2} : A + B \to LB$ is $[\bot \circ !_A, \mathsf{up}_B]$. For lists (either lazy or strict) we need a deconstructor for each (potential) element of the list, setting the deconstructor to $\bot$ when such an element does not exist. For strict lists, let $P_L = \{p_{L_n}\}_{n=1}^{\infty}$ where $p_{L_1} : \mathsf{List}\, A \to LA$ is hd and, for $n > 1$, $p_{L_n} = p_{L_{n-1}}{}^{\bot} \circ \mathsf{tl}$. Similarly, for lazy lists, let $P_{\mathrm{L}^z} = \{p_{\mathrm{L}^z_n}\}_{n=1}^{\infty}$ where $p_{\mathrm{L}^z_1} : \mathsf{Llist}\, A \to LA = \mathsf{lhd}$ and, for $n > 1$, $p_{\mathrm{L}^z_n} = p_{\mathrm{L}^z_{n-1}}{}^{\bot} \circ \mathsf{ltl}$.

Each deconstructor needs a related expression describing the deconstruction syntactically. For products there are constants describing the same function as the deconstructors themselves, so we have $e_{\times_1} = \mathtt{fst}(x)$ and $e_{\times_2} = \mathtt{snd}(x)$. Expressions of sum type are deconstructed with the `case` constant so we must use that to form the related expressions. Therefore let

$$e_{+_1} = \mathtt{case}\ x\ \mathtt{of}\ \mathtt{left} : \mathtt{lam}\ y.y\ \mathtt{right} : \mathtt{rec}\ z.z$$

and

$$e_{+_2} = \mathtt{case}\ x\ \mathtt{of}\ \mathtt{left} : \mathtt{rec}\ z.z\ \mathtt{right} : \mathtt{lam}\ y.y$$

For either lazy or strict lists, while the deconstructors do not match the constants we defined for FL, the morphisms we used to build the deconstructors do. Therefore we can let $e_{L_1} = \mathtt{head}(x)$ and, for $n > 1$, $e_{L_n} = e_{L_{n-1}}(\mathtt{tail}(x)) \equiv \mathtt{head}(\mathtt{tail}^{(n-1)}(x))$. These expressions are sufficient for both types of list.

The proofs that the constants are related to the constructed data types follows two steps: First, we interpret the general definition of $\precsim^{\mathrm{V}}_{\delta(\tau_1,\ldots,\tau_n)}$ and $\precsim^{\mathrm{N}}_{\delta(\tau_1,\ldots,\tau_n)}$ for a specific data type, giving a more concise method for determining if the relation holds. We then apply that method to show that each constant is adequate.

Appendix A contains the details of the proofs plus the proofs of adequacy for the ground type constants. In the rest of this section we show that the general definition for the adequacy of sum types is equivalent to what we would define for adequacy had we lacked a general method. Thus if we were to define $\precsim^{\mathrm{V}}_{\tau_1+\tau_2}$ without recourse to a general definition, we would then say that $z \precsim^{\mathrm{V}}_{\tau_1+\tau_2} v$ if and only if one of the following holds:

- $z = \bot$,
- For some $y : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!]$, $z = \mathsf{up} \circ \mathsf{inl} \circ y$, $v = \mathtt{inl}(v')$, and $\mathsf{up} \circ y \precsim^{\mathrm{V}}_{\tau_1} v'$, or
- For some $y : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!]$, $z = \mathsf{up} \circ \mathsf{inr} \circ y$, $v = \mathtt{inr}(v')$, and $\mathsf{up} \circ y \precsim^{\mathrm{V}}_{\tau_2} v'$.

We will show that the general definition holds precisely when the above conditions hold.

Suppose that $z \precsim^{\mathrm{V}}_{\tau_1+\tau_2} v$. Then by definition

$$L(!+!) \circ z \le L(!+!) \circ \mathcal{V}_{\mathrm{E}}[\![v]\!]\tau_1 + \tau_2,\ \ p_{+_1}{}^{\bot} \circ z \precsim^{\mathrm{V}}_{\tau_1} [v/x]e_{+_1},\ \text{ and } p_{+_2}{}^{\bot} \circ z \precsim^{\mathrm{V}}_{\tau_2} [v/x]e_{+_2}$$

As $z : \mathbf{1} \to L(\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!] + \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_2]\!])$ in $\mathbf{PDom}$, we know that there are three possibilities for z:

1. $z = \perp$ (all properties hold trivially).

2. $z = \mathsf{up} \circ \mathsf{inl} \circ y$. Then $L(!+!) \circ z = \mathsf{up}_{1+1} \circ \mathsf{inl}$. The value v can only be of the form $\mathtt{inl}(v')$ or $\mathtt{inr}(v')$ for some v'. If the latter, then $\mathcal{V}_{\mathrm{E}}[\![v : \tau_1 + \tau_2]\!] = \mathsf{up} \circ \mathsf{inr} \circ y'$, where $\mathcal{V}_{\mathrm{E}}[\![v']\!] = \mathsf{up} \circ y'$. This is inconsistent with the knowledge that L(!+!) $\circ\, z = \mathsf{up} \circ \mathsf{inl}$, so v must be of the form $v = \mathtt{inl}(v')$ for some closed value v' of type τ_1. Furthermore,

$$\mathsf{up} \circ x = [\mathsf{up}, \perp \circ\, !] \circ \mathsf{inl} \circ x = {p_{+_1}}^{\perp} \circ z \precsim^{\mathrm{V}}_{\tau_1} [v/x]e_{+_1}$$

and

$$[v/x]e_{+_1} = \mathtt{case}\ \mathtt{inl}(v')\ \mathtt{of}\ \mathtt{left} : \mathtt{lam}\ y.y\ \mathtt{right} : \mathtt{rec}\ z.z$$

so $\mathsf{up} \circ x \precsim^{\mathrm{V}}_{\tau_1} \mathtt{case}\ \mathtt{inl}(v')\ \mathtt{of}\ \mathtt{left} : \mathtt{lam}\ y.y\ \mathtt{right} : \mathtt{rec}\ z.z$. Lastly, clearly

$$\mathtt{case}\ \mathtt{inl}(v')\ \mathtt{of}\ \mathtt{left} : \mathtt{lam}\ y.y\ \mathtt{right} : \mathtt{rec}\ z.z \Rightarrow_{\mathrm{v}} v'$$

so $\mathsf{up} \circ x \precsim^{\mathrm{V}}_{\tau+1} v'$ as well.

3. $z = \mathsf{up} \circ \mathsf{inr} \circ y$. Using similar arguments as the previous case, we can show that v must have the form $\mathtt{inr}(v')$ for some value v' and $\mathsf{up} \circ y \precsim^{\mathrm{V}}_{\tau_2} v'$.

Now suppose that the three conditions stated earlier hold. If $z = \perp$ then $z \precsim^{\mathrm{V}}_{\tau_1+\tau_2} v$ trivially. Suppose that $z = \mathsf{up} \circ \mathsf{inl} \circ y$, $v = \mathtt{inl}(v')$ and $\mathsf{up} \circ y \precsim^{\mathrm{V}}_{\tau_1} v'$. Then we know that ${p_{+_1}}^{\perp} \circ z \precsim^{\mathrm{V}}_{\tau_1} [v/x]e_{+_1}$. Furthermore,

$$L(!+!) \circ z = \mathsf{up} \circ \mathsf{inl} = L(!+!) \circ \mathcal{V}_{\mathrm{E}}[\![v : \tau_1 + \tau_2]\!])$$

Finally,

$${p_{+_2}}^{\perp} \circ z = [\perp \circ\, !, \mathsf{up}] \circ \mathsf{inl} \circ y = \perp$$

so

$${p_{+_2}}^{\perp} \circ z \precsim^{\mathrm{V}}_{\tau_2} \mathtt{case}\ \mathtt{inl}(v)\ \mathtt{of}\ \mathtt{left} : \mathtt{rec}\ z.z\ \mathtt{right} : \mathtt{lam}\ y.y \equiv [v/x]e_{+_2}$$

Therefore $z \precsim^{\mathrm{V}}_{\tau_1+\tau_2} v$.

Using similar arguments we can show that if $z = \mathsf{up} \circ \mathsf{inr} \circ x$, $v = \mathtt{inr}(v')$ and $\mathsf{up} \circ x \precsim^{\mathrm{V}}_{\tau_2} v'$ then $z \precsim^{\mathrm{V}}_{\tau_1+\tau_2} v$ as well.

For call-by-name, given a morphism $z : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 + \tau_2]\!]$ and a closed value v of type $\tau_1 + \tau_2$, we can show that $z \precsim^{\mathrm{N}}_{\tau_1+\tau_2} v$ if and only if one of the following holds:

- $z = \perp$,
- For some $z' : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$ and closed expression e of type τ_1, $z = \mathsf{up} \circ \mathsf{inl} \circ z'$, $v = \mathtt{inl}(e)$, and $z' \precsim^{\mathrm{N}}_{\tau_1} e$.
- For some $z' : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]$ and closed expression e of type τ_2, $z = \mathsf{up} \circ \mathsf{inr} \circ z'$, $v = \mathtt{inr}(e)$, and $z' \precsim^{\mathrm{N}}_{\tau_2} e$.

The proof that this holds is similar to the previous proof.

Chapter 3

Call-by-value

3.1 Introduction

In this chapter we add cost to the call-by-value semantics. An evaluation strategy is called *call-by-value* if the input to functions is always evaluated before the function itself is called; i.e., the function operates solely on the *values* of the expressions. For example, when evaluating an expression such as $(\texttt{lam}\ x.e)(3+3)$ the expression $3+3$ is evaluated first, resulting in the value 6, then e is evaluated, with x bound to 6. The earliest functional languages (Lisp) and many current ones (Scheme, SML) are implemented as call-by-value even though the λ-calculus is frequently defined with a call-by-name semantics. There are several reasons for the preference, but the primary one is that call-by-value is easier to implement than call-by-need and usually more efficient than call-by-name. To handle most cases in which we must avoid evaluating part of an expression (such as with $\texttt{if}\ (\texttt{nil?}(l))\ \texttt{then}\ 0\ \texttt{else}\ \texttt{head}(l)$), call-by-value languages add primitive constructs, not only `if` as we do here, but multi-part case statements, pattern matching, and the use of non-strict versions of `or` and `and` as special constructs.

Another reason why call-by-value languages are frequently used is that analysis of the complexity of programs (without higher-order types) is easier than call-by-name, because we always know which parts of a program are being evaluated. For this reason, in this dissertation we add cost to the call-by-value semantics before the call-by-name. Even though adding costs to the operational semantics is straightforward, we can be reassured to find that the complexity analysis of programs derived from these costs is consistent with what we would expect to find from other forms of analysis. Therefore if we use similar techniques to add costs to the call-by-name semantics we can be confident that the costs derived reflect actual costs.

In this chapter we first add cost to the operational semantics defined in the previous chapter. The addition is straightforward and provides a basis for judging the addition of cost to the denotational semantics. We use the cost-equipped operational semantics to calculate the costs of a small example.

In section 3.3 we devise a *cost structure* in category theory that defines the way cost is added to the denotational semantics. We begin with both a standard technique for adding costs and our knowledge of monads, and then derive the necessary requirements for a sound, adequate, and separable denotational semantics. The cost structure not only allows us to add cost to the meaning of the program, but will also give us a function, C, on objects that allows us to clearly distinguish between *denotational values*, i.e., data that has no computational component (such as the integer 2) and *computations*, i.e., results of computing expressions. This distinction is similar to the one made by the lifting monad in the previous chapter.

In section 3.3.1 we show how the cost structure can be used to derive an intensional denotational semantics from the extensional one from Chapter 2. We also give a functor E that smoothly converts the intensional semantics back to the extensional one. In section 3.4 we prove that the intensional denotational semantics is sound. We also show that the meanings of the results of operational evaluation are closely related to denotational values.

In section 3.5 we show that the intensional semantics is adequate. By proving adequacy and soundness we not only show that the denotational semantics closely matches the operational semantics, but we also show that the meaning of a closed expression has one of only two simple forms. With this knowledge we are better able to examine specific examples as we can then make a number of simplifications not otherwise possible.

When we define cost structures in section 3.3 we do not list any non-trivial examples. In section 3.6 we do so. We first create another structure, called an *arrow cost structure*, that is simpler than the standard cost structure but which can still be used to derive a cost structure by using a type of category called an *arrow category*. In section 3.6.3 we show that an arrow cost structure exists in **PDom**.

In section 3.7 we give meanings to the constants in FL and show that they satisfy all the necessary assumptions. We then use these constants in section 3.8 to calculate the meanings of several simple examples. We also derive abbreviations for the meaning of a number of expressions so that we can clearly see the meaning of an expression using notation designed for that particular problem instead of notation based on standard categorical structures.

In section 3.9 we introduce an ordering on costs and extend it to an ordering on the meanings of programs. This ordering can be used to show that one program is faster than another, or that a program transformation improves speed. We show that this ordering is compositional and it relates well to an operationally defined ordering.

In section 3.10 we summarize the results of this chapter.

3.2 Call-by-value operational semantics

Figure 3.1 lists the intensional operational semantics. It is closely related to the semantics listed in chapter 2 with the main difference that both the evaluation judgment, $e \overset{t}{\Rightarrow}_{\mathrm{v}} v$, and the constant application judgment, $\mathrm{vapply}(c, v_1 \ldots, v_n) \overset{t}{\Rightarrow} v$, include a cost t. This element refers to the cost of the operation; we assume that it is an element of some set T of costs. T always contains the identity element 0 and has a binary operator $+$ such that $\langle T, 0, + \rangle$ forms a monoid. The symbols t_{true}, t_{false}, t_{app}, and t_{rec} also represent elements of T, not necessarily distinct from each other or from 0. We do not assume that $+$ is commutative; we can prove soundness and adequacy without that assumption. For simplicity, however, we do usually assume that $+$ is commutative when examining examples, thus writing $2t_{\mathrm{app}} + t_{\mathrm{false}}$ instead of $t_{\mathrm{app}} + t_{\mathrm{false}} + t_{\mathrm{app}}$.

There is no added cost when evaluating constants or abstractions because syntactic values such as these do not require further evaluation. Logically, values represent the final result of an evaluation so there should be no need to evaluate them again. For the same reason no additional cost is added when applying constants of arity greater than 0. If necessary, one can add one-time costs to programmatic "values" such as abstractions with the use of a special constant, such as `abs` of arity 1, with the application rule

$$\mathrm{vapply}(\mathtt{abs}, v) \overset{t_{\mathtt{abs}}}{\Rightarrow} v$$

If we limit initial programs so that `lam` $x.e$ is not allowed except when enclosed by an `abs` (making `abs` equivalent to the `fun` construct in ML), each initial use of an abstraction would add the cost

$$c \overset{0}{\Rightarrow}_{\mathrm{v}} c \qquad\qquad \mathtt{lam}\ x.e \overset{0}{\Rightarrow}_{\mathrm{v}} \mathtt{lam}\ x.e$$

$$\frac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{v}} c v_1 \ldots v_i \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{v}} v_{i+1} \quad \mathrm{vapply}(c, v_1, \ldots, v_{i+1}) \overset{t'}{\Rightarrow} v}{e_1(e_2) \overset{t'+t_2+t_1}{\Rightarrow}_{\mathrm{v}} v} \qquad \frac{[\mathtt{rec}\ x.e/x]e \overset{t}{\Rightarrow}_{\mathrm{v}} v}{\mathtt{rec}\ x.e \overset{t+t_{\mathrm{rec}}}{\Rightarrow}_{\mathrm{v}} v}$$

$$\frac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{v}} \mathtt{true} \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{v}} v}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \overset{t_2+t_{\mathrm{true}}+t_1}{\Rightarrow}_{\mathrm{v}} v} \qquad \frac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{v}} \mathtt{false} \quad e_3 \overset{t_3}{\Rightarrow}_{\mathrm{v}} v}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \overset{t_3+t_{\mathrm{false}}+t_1}{\Rightarrow}_{\mathrm{v}} v}$$

$$\frac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{v}} \mathtt{lam}\ x.e' \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{v}} v' \quad [v'/x]e' \overset{t'}{\Rightarrow}_{\mathrm{v}} v}{e_1(e_2) \overset{t'+t_{\mathrm{app}}+t_2+t_1}{\Rightarrow}_{\mathrm{v}} v}$$

Figure 3.1: Operational semantics for a call-by-value functional language

$$\frac{i < \mathrm{ar}(c)}{\mathrm{vapply}(c, v_1, \ldots, v_i) \overset{0}{\Rightarrow} c v_1 \ldots v_i} \qquad \frac{n = \mathrm{ar}(c) \quad c \in \mathbf{Construct}}{\mathrm{vapply}(c, v_1, \ldots, v_n) \overset{0}{\Rightarrow} c v_1 \ldots v_n}$$

Figure 3.2: Known rules for $\mathrm{vapply}(c, v_1, \ldots, v_n) \overset{t}{\Rightarrow} v$

of t_{abs}, but further evaluations of that abstraction would add 0 cost. Similar constructions can also be used to add a one-time cost to other values. Thus even though abstractions and other syntactic values have 0 cost it is possible to track the creation of such values.

Figure 3.2 lists the rules for constants that are known to always apply. These include rules for constructors, which form values and have no additional effect, and rules for insufficiently applied constants (such as `pair` 3). With all constants, we assume that no "work" (apart from evaluating the arguments themselves) occurs until all the arguments are present. This assumption is consistent with both an intuitive concept of the application of such constants and other definitions of cost, as in [37].

As the addition of cost does not alter the result of evaluation, the proofs that the intensional operational semantics is value and type sound are identical to the proofs for the extensional semantics.

3.2.1 Operational Example

By calculating the complexity of the following simple recursive program, we illustrate that the costs added to the operational semantics are reasonable:

$$\mathtt{length} = \mathtt{rec\ len.lam\ l.if\ nil?(l)\ then}\ \overline{0}\ \mathtt{else}\ \overline{1} + \mathtt{len(tail(l))}$$

The program calculates the length of a list.

To properly determine the cost of applying `length` we need an alternate method for displaying natural semantics; the layout used so far becomes confusing when the derivation is complex. Instead we use a form devised by [20]. Rather than write $e \overset{t}{\Rightarrow}_{\mathrm{v}} v$ we write

$$\begin{array}{l} e : \\ [t] \Rightarrow_{\mathrm{v}} v \end{array}$$

$$
\begin{array}{ll}
\text{vapply}(+, \overline{n}, \overline{m}) \stackrel{t_+}{\Rightarrow} \overline{n+m} & \text{vapply}(\texttt{tail}, v_1{::}v_2) \stackrel{t_{\text{tail}}}{\Rightarrow} v_2 \\
\text{vapply}(\texttt{nil?}, \texttt{nil}) \stackrel{t_{\text{nil?}}}{\Rightarrow} \texttt{true} & \text{vapply}(\texttt{nil?}, v_1{::}v_2) \stackrel{t_{\text{nil?}}}{\Rightarrow} \texttt{false}
\end{array}
$$

Figure 3.3: FL constants needed for `length`

Sub-derivations are entered in the middle, indented and marked with brackets, e.g., one of the conditional rules would be displayed as

$$
\begin{array}{l}
\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \\
\quad \left[\begin{array}{l} e_1 : \\ [t_1] \Rightarrow_{\text{v}} \texttt{true} \end{array} \right. \\
\quad \left[\begin{array}{l} e_2 : \\ [t_2] \Rightarrow_{\text{v}} v_2 \end{array} \right. \\
[t_1 + t_2 + t_{\text{if}}] \Rightarrow_{\text{v}} v_2
\end{array}
$$

instead of

$$
\frac{e_1 \stackrel{t_1}{\Rightarrow}_{\text{v}} \texttt{true} \quad e_2 \stackrel{t_2}{\Rightarrow}_{\text{v}} v_2}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \stackrel{t_1+t_2+t_{\text{if}}}{\Rightarrow}_{\text{v}} v_2}
$$

We can further simplify the layout to improve clarity. Dots (...) replace parts of expressions when they are both understood and not needed to determine the next step in the evaluation. Furthermore, we simply list any evaluation that has been previously derived without rederiving it. We also display short evaluations with no subparts using the standard notation $e \stackrel{t}{\Rightarrow}_{\text{v}} v$.

Before calculating the cost of applying `length` we also need to specify the rules for the integer and list constants used in the program. Figure 3.3 contains the application rules for the non-constructor constants. The rules are the same as those given in Chapter 2 except that in each case an additional constant cost (t_+, $t_{\text{nil?}}$, and t_{tail}) is included.

We start with the simplest case, evaluating `length(nil)`:

$$
\begin{array}{l}
\texttt{length(nil)} : \\
\quad \left[\begin{array}{l}
\texttt{length} \equiv \texttt{rec len.lam l.if} \ldots : \\
\quad \left[\begin{array}{l} \texttt{lam l.if nil?(l) then } \overline{0} \texttt{ else } \overline{1} + \texttt{length(tail(l))} : \\ [0] \Rightarrow_{\text{v}} \texttt{lam l.if} \ldots \end{array} \right. \\
[t_{\text{rec}}] \Rightarrow_{\text{v}} \texttt{lam l.if} \ldots
\end{array} \right. \\
\quad \left[\texttt{nil} \stackrel{0}{\Rightarrow}_{\text{v}} \texttt{nil} \right. \\
\quad \left[\begin{array}{l}
\texttt{if nil?(nil) then } \overline{0} \texttt{ else } \overline{1} + \texttt{length(tail(nil))} : \\
\quad \left[\begin{array}{l}
\texttt{nil?(nil)} : \\
\quad \left[\texttt{nil?} \stackrel{0}{\Rightarrow}_{\text{v}} \texttt{nil?} \right. \\
\quad \left[\texttt{nil} \stackrel{0}{\Rightarrow}_{\text{v}} \texttt{nil} \right. \\
\quad \left[\text{vapply}(\texttt{nil?}, \texttt{nil}) \stackrel{t_{\text{nil?}}}{\Rightarrow} \texttt{true} \right. \\
[t_{\text{nil?}}] \Rightarrow_{\text{v}} \texttt{true}
\end{array} \right. \\
\quad \left[\overline{0} \stackrel{0}{\Rightarrow}_{\text{v}} \overline{0} \right. \\
[t_{\text{true}} + t_{\text{nil?}}] \Rightarrow_{\text{v}} \overline{0}
\end{array} \right. \\
[t_{\text{true}} + t_{\text{nil?}} + t_{\text{app}} + t_{\text{rec}}] \Rightarrow_{\text{v}} \overline{0}
\end{array}
$$

If e is any value such that $e \overset{t}{\Rightarrow}_{\mathrm{n}} \mathtt{nil}$, then evaluating $\mathtt{length}(e)$ is much the same:

$$
\begin{array}{l}
\mathtt{length}(e) : \\
\quad \mathtt{length} \overset{t_{\mathrm{rec}}}{\Rightarrow}_{\mathrm{v}} \mathtt{lam\ l.if} \ldots \\
\quad e \overset{t}{\Rightarrow}_{\mathrm{v}} \mathtt{nil} \\
\qquad \mathtt{if\ nil?(nil)\ then}\ \overline{0}\ \mathtt{else}\ \overline{1} + \mathtt{length(tail(nil))} : \\
\qquad [t_{\mathrm{true}} + t_{\mathrm{nil?}}] \Rightarrow_{\mathrm{v}} \overline{0} \\
[t_{\mathrm{true}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}} + t] \Rightarrow_{\mathrm{v}} \overline{0}
\end{array}
$$

We thus have the following *derived* rule which we can use in future examples:

$$
\frac{e \overset{t}{\Rightarrow}_{\mathrm{v}} \mathtt{nil}}{\mathtt{length}(e) \overset{t_{\mathrm{true}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}} + t}{\Rightarrow_{\mathrm{v}}} \overline{0}}
$$

Our next example it also very simple; it computes the length of a list of the form $v::\mathtt{nil}$. In this example we also omit evaluations such as $c \overset{0}{\Rightarrow}_{\mathrm{v}} c$ when obvious.

$$
\begin{array}{l}
\mathtt{length}(v::\mathtt{nil}) : \\
\quad \mathtt{length} \overset{t_{\mathrm{rec}}}{\Rightarrow}_{\mathrm{v}} \mathtt{lam\ l.if} \ldots \\
\qquad \mathtt{if\ nil?}(v::\mathtt{nil})\ \mathtt{then}\ \overline{0}\ \mathtt{else}\ \overline{1} + \mathtt{length(tail}(v::\mathtt{nil})) : \\
\qquad\quad \mathtt{nil?}(v::\mathtt{nil}) : \\
\qquad\qquad \mathrm{vapply}(\mathtt{nil?}, v::\mathtt{nil}) \overset{t_{\mathrm{nil?}}}{\Rightarrow} \mathtt{false} \\
\qquad\quad [t_{\mathrm{nil?}}] \Rightarrow_{\mathrm{v}} \mathtt{false} \\
\qquad\quad \overline{1} + \mathtt{length(tail}(v::\mathtt{nil})) : \\
\qquad\qquad \mathtt{length(tail}(v::\mathtt{nil})) : \\
\qquad\qquad\quad \mathtt{tail}(v::\mathtt{nil}) : \\
\qquad\qquad\qquad \mathrm{vapply}(\mathtt{tail}, v::\mathtt{nil}) \overset{t_{\mathrm{tail}}}{\Rightarrow} \mathtt{nil} \\
\qquad\qquad\quad [t_{\mathrm{tail}}] \Rightarrow_{\mathrm{v}} \mathtt{nil} \\
\qquad\qquad [t_{\mathrm{true}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{tail}}] \Rightarrow_{\mathrm{v}} \overline{0} \\
\qquad\qquad \mathrm{vapply}(+, \overline{1}, \overline{0}) \overset{t_{+}}{\Rightarrow} \overline{1} \\
\qquad\quad [t_{+} + t_{\mathrm{true}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{tail}}] \Rightarrow_{\mathrm{v}} \overline{1} \\
\qquad [t_{+} + t_{\mathrm{true}} + 2t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{tail}} + t_{\mathrm{false}}] \Rightarrow_{\mathrm{v}} \overline{1} \\
[t_{+} + t_{\mathrm{true}} + 2t_{\mathrm{nil?}} + 2t_{\mathrm{app}} + 2t_{\mathrm{rec}} + t_{\mathrm{tail}} + t_{\mathrm{false}}] \Rightarrow_{\mathrm{v}} \overline{1}
\end{array}
$$

To find the overall complexity, we need to know the cost of applying `length` to lists in general. We can determine this cost by induction on the length of the list:

Theorem 3.2.1 *Suppose that* $e \overset{t}{\Rightarrow}_{\mathrm{v}} [v_1, \ldots, v_n]$*. Then* $\mathtt{length}(e) \overset{t'}{\Rightarrow}_{\mathrm{v}} \overline{n}$*, where*

$$t' = n(t_{+} + t_{\mathrm{tail}} + t_{\mathrm{false}}) + (n+1)(t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}) + t_{\mathrm{true}} + t$$

Proof. By induction on n. We have already shown the cases where $n = 0$ and where $n = 1$. Suppose that $n > 0$ and that the theorem holds for $n-1$. Then

$$
\begin{array}{l}
\texttt{length}(e) : \\
\quad \left[\texttt{length} \overset{t_{\mathrm{rec}}}{\Rightarrow}_{\mathrm{v}} \texttt{lam}\ l.\texttt{if} \ldots \right. \\
\quad \left[e \overset{t}{\Rightarrow}_{\mathrm{v}} [v_1, \ldots, v_n] \right. \\
\quad\quad \texttt{if nil?}([v_1, \ldots, v_n])\ \texttt{then}\ \overline{0}\ \texttt{else}\ \overline{1} + \texttt{length}(\texttt{tail}([v_1, \ldots, v_n])) : \\
\quad\quad\quad \texttt{nil?}([v_1, \ldots, v_n]) : \\
\quad\quad\quad\quad \left[\mathrm{vapply}(\texttt{nil?}, [v_1, \ldots, v_n]) \overset{t_{\mathrm{nil?}}}{\Rightarrow} \texttt{false} \right. \\
\quad\quad\quad [t_{\mathrm{nil?}}] \Rightarrow_{\mathrm{v}} \texttt{false} \\
\quad\quad\quad \overline{1} + \texttt{length}(\texttt{tail}([v_1, \ldots, v_n])) : \\
\quad\quad\quad\quad \texttt{length}(\texttt{tail}([v_1, \ldots, v_n])) : \\
\quad\quad\quad\quad\quad \texttt{tail}([v_1, \ldots, v_n]) : \\
\quad\quad\quad\quad\quad\quad \left[\mathrm{vapply}(\texttt{tail}, [v_1, \ldots, v_n]) \overset{t_{\mathrm{tail}}}{\Rightarrow} [v_2, \ldots, v_n] \right. \\
\quad\quad\quad\quad\quad [t_{\mathrm{tail}}] \Rightarrow_{\mathrm{v}} [v_2, \ldots, v_n] \\
\quad\quad\quad\quad [(n-1)(t_+ + t_{\mathrm{tail}} + t_{\mathrm{false}}) + n(t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}) + t_{\mathrm{true}} + t_{\mathrm{tail}}] \Rightarrow_{\mathrm{v}} \overline{n-1} \quad * \\
\quad\quad\quad\quad \left[\mathrm{vapply}(+, \overline{1}, \overline{n-1}) \overset{t_+}{\Rightarrow} \overline{n} \right. \\
\quad\quad\quad [(n-1)(t_{\mathrm{false}}) + n(t_+ + t_{\mathrm{tail}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}) + t_{\mathrm{true}}] \Rightarrow_{\mathrm{v}} \overline{n} \\
\quad\quad [n(t_{\mathrm{false}} + t_+ + t_{\mathrm{tail}} + t_{\mathrm{app}} + t_{\mathrm{rec}}) + (n+1)t_{\mathrm{nil?}} + + t_{\mathrm{true}}] \Rightarrow_{\mathrm{v}} \overline{n} \\
[n(t_{\mathrm{false}} + t_+ + t_{\mathrm{tail}}) + (n+1)(t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}) + + t_{\mathrm{true}} + t] \Rightarrow_{\mathrm{v}} \overline{n}
\end{array}
$$

The spot marked with an $(*)$ indicates where we use the induction hypothesis. $\square$

From this theorem we find that the cost of applying `length` is proportional to the length of the input list. This result is consistent with other standard approaches to determining the complexity of `length`.

3.3 Adding cost to the denotational semantics

In the extensional denotational semantics from Chapter 2, the meaning of a well typed expression was a morphism from an object A_0, representing the environment, to an object D representing the possible results. One method for adding cost is to change the meaning of an expression so that it is a morphism from A_0 to an object $D \times \mathbf{T}$, where $\mathbf{T}$ is some object representing the cost of evaluation. The resulting semantics, however, is rather complex. For example, the meaning of an application $e_1(e_2)$ might look like

$$(\mathsf{id} \times \mathsf{m}) \circ \alpha^r \circ (\mathsf{app} \times \mathsf{m}) \circ \phi \circ \langle f_1, f_2 \rangle$$

where f_1 and f_2 are the meanings of e_1 and e_2, respectively, $\mathsf{m} : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$ combines costs, and α^r and ϕ are product isomorphisms as defined in Figure 2.10 of Chapter 2.

More importantly, if we only use $D \times \mathbf{T}$, we cannot easily change the semantics to include other possible structures that might work as well or better for the intensional semantics. Instead, we would prefer to find an abstract method for adding cost for which this method using products would be a valid example.

If we had chosen to use products, we would find, while defining the semantics, that we would frequently need certain general operations. To define a value, we would need to initialize an element so it had 0 cost (for example, the meaning of `true` would be $\langle \mathsf{tt}, 0 \rangle$). Also, if the value of an expression e was the result of evaluating a subexpression e', we would need to be able to take the meaning of e' and add possible additional costs accumulated while evaluating other parts of e plus constant costs created by the structure of e itself. For example, when evaluating $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$, if e_1 evaluates to $\texttt{true}$, then the meaning of $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ is the meaning of e_2 plus the cost of evaluating e_1 and the constant cost t_{true} for evaluating a (true) conditional. Lastly we would need to combine costs from completely separate subevaluations, most noticeably when evaluating pairs ($<e_1, e_2>$) but also when evaluating applications.

We can describe most of these operations using a strong monad. Furthermore, for any object A with certain properties (relating to the ability to add costs together as described in detail in section 3.6.3) the product functor $- \times A$ forms a strong monad. Therefore, it seems reasonable to begin with a strong monad (or, more precisely, a Kleisli triple with strength) as our cost structure.

Thus let C be a function from objects to objects. C adds the ability to handle cost to an object. To add an initial (0) cost, let $\eta_A : A \to CA$ for any object A. To compose multiple morphisms of the form $f : A \to CB$, we want to be able to "lift" f to a morphism from $CA \to CB$, so for each morphism $f : A \to CB$ let $f^* : CA \to CB$. Finally to combine costs directly, let $\psi_{A,B} : CA \times CB \to C(A \times B)$ for any objects A and B. Lastly we need the ability to add non-zero constant costs. Given the monoid T of costs defined in the previous section, for any $t \in T$ and any object A, let $[\![t]\!]_A : CA \to CA$. The morphism $[\![t]\!]$ adds the cost t while leaving everything else unchanged.

These elements enable us to define a sound intensional semantics, but not one that is necessarily separable or adequate. For that we require a method for recovering the extensional semantics from the intensional. Thus let E be a functor that converts the intensional semantics to the extensional. By making E a functor we do not require that the intensional and extensional semantics be in the same category. Let $\mathcal{C}^{\mathrm{E}}$ refer to the category of the extensional semantics and $\mathcal{C}^{\mathrm{I}}$ refer to the category of the intensional semantics. Then C, η, $(-)^*$, ψ, and $[\![-]\!]$ all operate in $\mathcal{C}^{\mathrm{I}}$, while E is a functor from $\mathcal{C}^{\mathrm{I}}$ to $\mathcal{C}^{\mathrm{E}}$.

We will refer to the data $(C, \eta, (-)^*, \psi, [\![-]\!], E)$ as a *cost structure* on T. We next must determine what properties are required for soundness, adequacy, and separability. In particular, it will be useful to know if not all properties of a strong monad are needed. By specifying a smaller set of required properties we have a more precise description of the process of adding cost.

We require only four properties of a cost structure in order to form a sound semantics, all of them involving $[\![-]\!]$. The first two properties formalize the concept that $[\![t]\!]$ adds cost t; we require that $[\![0]\!] = \mathsf{id}$ and that $[\![t_2]\!] \circ [\![t_1]\!] = [\![t_1 + t_2]\!]$.

The third property is that $f^* \circ [\![t]\!] \circ \eta = [\![t]\!] \circ f$. This property essentially states that the method of combining cost used by $(-)^*$ is to add the input cost after evaluating the input data. It also means that $[\![t]\!]$ adds cost differently than f^*. In particular, $[\![t]\!]$ does not necessarily equal $([\![t]\!] \circ \eta)^*$. When $t = 0$, the property turns into the Kleisli triple property that $f^* \circ \eta = f$.

The fourth property needed controls how ψ combines costs in comparison with $[\![t]\!]$. With the call-by-value semantics, if we choose the rule that $\psi \circ ([\![t_1]\!] \times [\![t_2]\!]) = [\![t_1 + t_2]\!] \circ \psi$, we find that $+$ must then be commutative. To see how the rule implies commutativity, note that

$$\begin{aligned} \psi \circ ([\![t_1]\!] \times [\![0]\!]) \circ ([\![0]\!] \times [\![t_2]\!]) &= [\![t_1]\!] \circ \psi \circ ([\![0]\!] \times [\![t_2]\!]) \\ &= [\![t_1]\!] \circ [\![t_2]\!] \circ \psi \\ &= [\![t_2 + t_1]\!] \circ \psi \end{aligned}$$

but also

$$\begin{aligned} \psi \circ ([\![t_1]\!] \times [\![0]\!]) \circ ([\![0]\!] \times [\![t_2]\!]) &= \psi \circ ([\![t_1]\!] \times [\![t_2]\!]) \\ &= [\![t_1 + t_2]\!] \circ \psi \end{aligned}$$

Using the rule that $\psi \circ (\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket) = \llbracket t_2 + t_1 \rrbracket \circ \psi$ instead does not solve the problem, as can be seen by reducing $\psi \circ (\llbracket 0 \rrbracket \times \llbracket t_1 \rrbracket) \circ (\llbracket t_2 \rrbracket \times \llbracket 0 \rrbracket)$ both ways.

If instead we require that $\psi \circ (\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket) \circ (\eta \times \eta)$ is either $\llbracket t_1 + t_2 \rrbracket \circ \eta$ or $\llbracket t_2 + t_1 \rrbracket \circ \eta$ we are forced to follow the second approach, removing the need to assume commutativity. Intuitively, the property states that ψ combines costs directly, but the property is limited to only those cases where we know the entire cost of the input.

We still need to decide whether to require that $\psi \circ (\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket) \circ (\eta \times \eta)$ equals $\llbracket t_1 + t_2 \rrbracket \circ \eta$ or $\llbracket t_2 + t_1 \rrbracket \circ \eta$. Both versions form sound semantics; we choose the latter form to better match the similar requirement that $\llbracket t_1 \rrbracket \circ \llbracket t_2 \rrbracket = \llbracket t_2 + t_1 \rrbracket$. With this choice, the order of evaluation can often be reflected by reading the costs from right to left. When the cost t is 0, the above property reduces to one of the properties of a strong monad (using ψ) seen in Figure 2.12.

These four properties are sufficiently limited so that C would not even need to be a functor. In practice, however, we should expect that any particular cost structure will be a strong monad simply because the requirements are very natural. In particular, we will be calling a cost structure *proper* if it also defines a monad and *strong* if it also defines a strong monad. All known examples of cost structures will be shown to be also strong.

To guarantee separability we need further requirements. If we require that $E\llbracket t \rrbracket_A = \mathsf{id}_{EA}$ for all objects A in $\mathcal{C}^{\mathrm{I}}$ and costs t, then we can apply E to generate a sound extensional semantics from the intensional. This requirement simply states that $\llbracket t \rrbracket$ has no extensional effect.

The extensional semantics, however, needs to be the same as the one developed in Chapter 2. Then we can use the adequacy of that semantics to prove adequacy of the intensional semantics. In section 3.3.1 we construct the intensional semantics by replacing the lifting monad with the equivalent constructs from the cost structure and add constant costs where needed. To recover the extensional semantics, therefore, we require that E be a representation, i.e., it preserves products, exponents, fixed points, and conditionals. Furthermore, we require that E applied to each element of $(C, \eta, (-)^*, \psi)$ gives us the lifting monad $(L, \mathsf{up}, (-)^{\perp}, \mathsf{smash})$.

To guarantee adequacy we need properties that deal with $\perp$. First, for any object A in $\mathcal{C}^{\mathrm{I}}$, CA must be strict. This property also ensures that fixed points are always well defined. Second, because non-terminating expressions are the same for both semantics, for any $f : \mathbf{1} \to CA$, we require that $f = \perp$ if and only if $Ef = \perp$ as well. Intuitively, this property states that there is no intensional cost for non-terminating expressions.

More generally, we need a property concerning ordering so that we can ensure that fixed points behave as expected. Suppose that $g, g' : \mathbf{1} \to CA$, and t, t' are costs such that $\llbracket t \rrbracket \circ \eta \circ g \leq \llbracket t' \rrbracket \circ \eta \circ g'$. We need to be able to say something about the relation of t and t' and of g and g'. In particular we want $g \leq g'$. For the ordering on costs, the relation $\leq$ is for fixed points, and thus refers to the amount of information about a meaning, not relative speed or amount of time taken so far. We either know the cost of an operation or do not; this implies that costs are discretely ordered. Rather than require that $t = t'$, however, we require that $\llbracket t \rrbracket = \llbracket t' \rrbracket$. This allows definition of $\llbracket - \rrbracket$ where some distinct costs are given the same meanings, such as a trivial cost function where $\llbracket t \rrbracket = \llbracket 0 \rrbracket$ for all costs t. Thus we require that if $\llbracket t \rrbracket \circ \eta \circ g \leq \llbracket t' \rrbracket \circ \eta \circ g'$ then $\llbracket t \rrbracket = \llbracket t' \rrbracket$ and $g \leq g'$.

Definition 3.3.1 Let T be a monoid set, $\mathcal{C}^{\mathrm{E}}$ be a Cartesian closed, **PDom**-enriched category with lifts and conditionals, and let $\mathcal{C}^{\mathrm{I}}$ be a Cartesian closed, **PDom**-enriched category with conditionals. Then a *cost structure of* T *on* $\mathcal{C}^{\mathrm{I}}$ *over* $\mathcal{C}^{\mathrm{E}}$ is a sextuple $(C, \eta, (-)^*, \psi, \llbracket - \rrbracket, E)$, where

- C is a function on objects of $\mathcal{C}^{\mathrm{I}}$
- For any object A in $\mathcal{C}^{\mathrm{I}}$, $\eta_A : A \to CA$

- For any morphism $f : A \to CB$, $f^* : CA \to CB$
- For any objects A, B in $\mathcal{C}^{\mathrm{I}}$, $\psi_{A,B} : CA \times CB \to C(A \times B)$
- For any object A in $\mathcal{C}^{\mathrm{I}}$ and any cost $t \in T$, $[\![t]\!]_A : CA \to CA$
- E is a functor from $\mathcal{C}^{\mathrm{I}}$ to $\mathcal{C}^{\mathrm{E}}$.

and the following properties hold:

Soundness Properties:

1. For all objects A
$$[\![0]\!]_A = \mathsf{id}_{CA}$$

2. For all objects A and all $t_1, t_2 \in T$,
$$[\![t_2]\!]_A \circ [\![t_1]\!]_A = [\![t_1 + t_2]\!]_A$$

3. For all $f : A \to CB$ and all $t \in T$,
$$f^* \circ [\![t]\!]_A \circ \eta_A = [\![t]\!]_B \circ f$$

4. For all $t_1, t_2 \in T$ and pairs of objects A_1 and A_2,
$$\psi_{A_1,A_2} \circ ([\![t_1]\!]_{A_1} \times [\![t_2]\!]_{A_2}) \circ (\eta_{A_1} \times \eta_{A_2}) = [\![t_2 + t_1]\!]_{A_1 \times A_2} \circ \eta_{A_1 \times A_2}$$

Separability properties:

1. E is a representation of Cartesian closed, **PDom**-enriched categories with conditionals.
2. $E(C, \eta, (-)^*, \psi)$ is the lifting monad, i.e., for all objects A of $\mathcal{C}^{\mathrm{I}}$, $E(CA) = L(EA)$, $E\eta_A = \mathsf{up}_{EA}$, $Ef^* = (Ef)^{\perp}$, and $E\psi_{A,B} = \mathsf{smash}_{EA,EB}$.
3. For all costs t, $E[\![t]\!]_A = \mathsf{id}_{EA}$.

Adequacy properties:

1. For all objects A, CA is strict.
2. For all $x : \mathbf{1} \to CA$, $x = \perp$ if and only if $Ex = \perp$.

Ordering property:

1. For all $z, z' : \mathbf{1} \to CA$ and all costs t, t', if $[\![t]\!] \circ \eta \circ z \leq [\![t']\!] \circ \eta \circ z'$, then $[\![t]\!] = [\![t']\!]$ and $z \leq z'$.

The category $\mathcal{C}^{\mathrm{E}}$ is the cost structure's *extensional category*, while $\mathcal{C}^{\mathrm{I}}$ is its *intensional category*.

$$\begin{tikzcd}
& CA \arrow[dr, "\llbracket t_2 \rrbracket"] & \\
CA \arrow[ur, "\llbracket t_1 \rrbracket"] \arrow[rr, "\llbracket t_1 + t_2 \rrbracket"'] & & CA
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "\eta"] \arrow[d, "f"'] & CA \arrow[r, "\llbracket t \rrbracket"] & CA \arrow[d, "f^*"] \\
CB \arrow[rr, "\llbracket t \rrbracket"'] & & CB
\end{tikzcd}$$

$$\begin{tikzcd}
A_1 \times A_2 \arrow[r, "\eta \times \eta"] \arrow[d, "\eta"'] & CA_1 \times CA_2 \arrow[r, "\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket"] & CA_1 \times CA_2 \arrow[d, "\psi"] \\
C(A_1 \times A_2) \arrow[rr, "\llbracket t_2 + t_1 \rrbracket"'] & & C(A_1 \times A_2)
\end{tikzcd}$$

Figure 3.4: Soundness properties of cost structures

Soundness properties 2-4 are shown as diagrams in Figure 3.4.

Definition 3.3.2 A cost structure $(C, \eta, (-)^*, \psi, \llbracket - \rrbracket, E)$ of T on $\mathcal{C}^{\mathrm{I}}$ over $\mathcal{C}^{\mathrm{E}}$ is *proper* if $(C, \eta, (-)^*)$ is a Kleisli triple on $\mathcal{C}^{\mathrm{I}}$, i.e., $(C, \eta, (\mathsf{id})^*)$ is a monad on $\mathcal{C}^{\mathrm{I}}$. The cost structure is *strong* if $(C, \eta, (\mathsf{id})^*, \tau)$ is a strong monad, where $\tau = \psi \circ (\mathsf{id} \times \eta)$.

As the soundness property 3 implies the first property of a Kleisli triple, to prove that a cost structure is proper we need only to show that the following holds:

- For all objects A,
$$(\eta_A)^* = \mathsf{id}_{CA}$$

- For all morphisms $f : A \to CB$ and $g : B \to CD$,
$$g^* \circ f^* = (g^* \circ f)^*$$

Similarly, as the soundness property 4 implies one of the strength properties of ψ and thus τ, to prove that a cost structure is strong we need to show that it is proper and the following holds:

- For all objects A and B,
$$C\pi_1 \circ \tau_{A,B} = \pi_1$$

- For all objects A, B, and D,
$$\tau_{A,B\times D} \circ \alpha^r_{CA,CB,CD} = C\alpha^r_{A,B,D} \circ \tau_{A\times B,D} \circ (\tau_{A,B} \times \mathsf{id}_{CD})$$

- For all morphisms $f : A_1 \to CB_1$ and $g : A_2 \to B_2$,

$$\tau_{B_1,B_2} \circ (f^* \times g) = (\tau_{B_1,B_2} \circ (f \times g))^* \circ \tau_{A_1,A_2}$$

We can also guarantee that a cost structure is strong with some relatively straightforward conditions on $\mathcal{C}^{\mathrm{I}}$ and C. A category is *well pointed* if for all morphisms $f, g : A \to B$, whenever for all $x : \mathbf{1} \to A$, $f \circ x = g \circ x$, then $f = g$.

Theorem 3.3.1 *Suppose that $\mathcal{C}^{\mathrm{I}}$ is well pointed, and that $(C, \eta, (-)^*, \psi, [\![-]\!], E)$ is a cost structure on $\mathcal{C}^{\mathrm{I}}$ over $\mathcal{C}^{\mathrm{E}}$. Further suppose that for any object X and any element $z : \mathbf{1} \to CX$, either $z = \bot$ or $z = [\![t]\!] \circ \eta \circ x$ for some $t \in T$ and $x : \mathbf{1} \to X$. Then $(C, \eta, (-)^*, \psi, [\![-]\!], E)$ is a strong cost structure.*

Proof. Before we can show that the cost structure is strong, we first note that the separability and adequacy properties give us the following properties of the cost structure and $\bot$:

- For all $f : A \to CB$
$$f^* \circ \bot_A = \bot_B$$
- For all costs t and objects A,
$$[\![t]\!]_A \circ \bot_A = \bot_A$$
- For all objects A_1 and A_2,
$$\psi_{A_1,A_2} \circ (\bot_{CA_1} \times \mathsf{id}_{CA_2}) = \bot_{C(A_1 \times A_2)} \circ !_{\mathbf{1} \times CA_2}$$
and
$$\psi_{A_1,A_2} \circ (\mathsf{id}_{CA_1} \times \bot_{CA_2}) = \bot_{C(A_1 \times A_2)} \circ !_{CA_1 \times \mathbf{1}}$$

For example, by the separability properties 1 and 2,

$$E(f^* \circ \bot) = (Ef)^{\bot} \circ \bot = \bot$$

thus by the adequacy property 2, $f^* \circ \bot_A$ must be $\bot$ as well.

We can now show that $(C, \eta, (-)^*)$ is a Kleisli triple. To do this we only need to show that the other two Kleisli triple properties hold. The proofs of both are by case analysis on morphisms from $\mathbf{1}$ to CA; we include proof that $g^* \circ f^* = (g^* \circ f)^*$.

- For $f : A \to CB$ and $g : B \to CD$, $g^* \circ f^* = (g^* \circ f)^*$

  Let $z : \mathbf{1} \to CA$. If $z = \bot$, then

$$g^* \circ f^* \circ z = \bot = (g^* \circ f)^* \circ z$$

  Otherwise, there exists a cost t and a morphism $x : \mathbf{1} \to A$ such that $z = [\![t]\!] \circ \eta \circ x$. Then

$$\begin{aligned} (g^* \circ f)^* \circ z = &\ (g^* \circ f)^* \circ [\![t]\!] \circ \eta \circ x \\ = &\ [\![t]\!] \circ g^* \circ f \circ x && \text{Soundness prop. 3} \end{aligned}$$

  and

$$\begin{aligned} g^* \circ f^* \circ z = &\ g^* \circ f^* \circ [\![t]\!] \circ \eta \circ x \\ = &\ g^* \circ [\![t]\!] \circ f \circ x \end{aligned}$$

Now $f \circ x : \mathbf{1} \to CB$. If $f \circ x = \perp$, then

$$[\![t]\!] \circ g^* \circ f \circ x = \perp = g^* \circ [\![t]\!] \circ f \circ x$$

Otherwise, there exists a cost t' and a morphism $x' : \mathbf{1} \to B$ such that $f \circ x = [\![t']\!] \circ \eta \circ x'$. Thus

$$\begin{array}{rcll} [\![t]\!] \circ g^* \circ f \circ x & = & [\![t]\!] \circ g^* \circ [\![t']\!] \circ \eta \circ x' & \\ & = & [\![t]\!] \circ [\![t']\!] \circ g \circ x' & \text{Soundness prop. 3} \\ & = & [\![t' + t]\!] \circ g \circ x' & \text{Soundness prop. 2} \\ & = & g^* \circ [\![t' + t]\!] \circ \eta \circ x' & \text{Soundness prop. 3} \\ & = & g^* \circ [\![t]\!] \circ [\![t']\!] \circ \eta \circ x' & \text{Soundness prop. 2} \\ & = & g^* \circ [\![t]\!] \circ f \circ x & \end{array}$$

Therefore $(g^* \circ f)^* \circ z = g^* \circ f^* \circ z$, so $(g^* \circ f)^* = g^* \circ f^*$.

We next show that the three extra strength properties are satisfied. Again, the proofs are similar to the one above; we only list one of the properties.

- $C\alpha^r_{A_1,A_2,A_3} \circ \tau_{A_1 \times A_2, A_3} \circ (\tau_{A_1,A_2} \times \mathsf{id}_{A_3}) = \tau_{A_1, A_2 \times A_3} \circ \alpha^r_{CA_1,A_2,A_3}$

  Let $z : \mathbf{1} \to (CA_1 \times A_2) \times A_3$. Then $z = \langle\langle z_1, x_2\rangle, x_3\rangle$, where $z_1 : \mathbf{1} \to CA_1$, $x_2 : \mathbf{1} \to A_2$ and $x_3 : \mathbf{1} \to A_3$. Suppose that $z_1 = \perp$. Then as noted

  $$\psi \circ (\perp \times \mathsf{id}) = \perp \circ\, !$$

  so for any morphism $y : \mathbf{1} \to CA$,

  $$\psi \circ \langle \perp, g\rangle = \perp$$

  Therefore

  $$\begin{array}{rl} & C\alpha^r \circ \tau \circ (\tau \times \mathsf{id}) \circ z \\ = & C\alpha^r \circ \psi \circ (\mathsf{id} \times \eta) \circ (\psi \times \mathsf{id}) \circ \langle\langle \perp, \eta \circ x_2\rangle, x_3\rangle \\ = & C\alpha^r \circ \psi \circ (\mathsf{id} \times \eta) \circ \langle \perp, x_3\rangle \\ = & C\alpha^r \circ \psi \circ \langle \perp, \eta \circ x_3\rangle \\ = & C\alpha^r \circ \perp \\ = & \perp \\ = & \psi \circ \langle \perp, \eta \circ \langle x_2, x_3\rangle\rangle \\ = & \tau \circ \langle \perp, \langle x_2, x_3\rangle\rangle \\ = & \tau \circ \alpha^r \circ z \end{array}$$

  Otherwise, there exists a cost t and a morphism $x_1 : \mathbf{1} \to A_1$ such that $z_1 = [\![t]\!] \circ \eta \circ x_1$. Therefore

  $$\begin{array}{rll} & C\alpha^r \circ \tau \circ (\tau \times \mathsf{id}) \circ z & \\ = & C\alpha^r \circ \psi \circ (\mathsf{id} \times \eta) \circ (\psi \times \mathsf{id}) \circ \langle\langle [\![t]\!] \circ \eta \circ x_1, \eta \circ x_2\rangle, x_3\rangle & \\ = & C\alpha^r \circ \psi \circ (\mathsf{id} \times \eta) \circ \langle [\![t]\!] \circ \eta \circ \langle x_1, x_2\rangle, x_3\rangle & \\ = & C\alpha^r \circ \psi \circ \langle [\![t]\!] \circ \eta \circ \langle x_1, x_2\rangle, \eta \circ x_3\rangle & \\ = & C\alpha^r \circ [\![t]\!] \circ \eta \circ \langle\langle x_1, x_2\rangle, x_3\rangle & \text{Soundness prop. 4} \\ = & [\![t]\!] \circ \eta \circ \alpha^r \circ \langle\langle x_1, x_2\rangle, x_3\rangle & \text{Soundness prop. 3} \\ = & [\![t]\!] \circ \eta \circ \langle x_1, \langle x_2, x_3\rangle\rangle & \text{Soundness prop. 4} \\ = & \psi \circ ([\![t]\!] \times \eta) \circ (\eta \times \mathsf{id}) \circ \langle x_1, \langle x_2, x_3\rangle\rangle & \\ = & \tau \circ \langle [\![t]\!] \circ \eta \circ x_1, \langle x_2, x_3\rangle\rangle & \\ = & \tau \circ \alpha^r \circ \langle\langle [\![t]\!] \circ \eta \circ x_1, x_2\rangle, x_3\rangle & \\ = & \tau \circ \alpha^r \circ z & \end{array}$$

  Therefore $C\alpha^r \circ \tau \circ (\tau \times \mathsf{id}) = \tau \circ \alpha^r$.

□

For any category $\mathcal{C}$ that is Cartesian closed, **PDom** enriched, and has lifts, we can define a trivial strong cost structure $(L, \mathsf{up}, (-)^{\perp}, \mathsf{smash}, \mathsf{id}, I)$ on $\mathcal{C}$ over $\mathcal{C}$, where all costs equal 0. In section 3.6.3 we present a non-trivial example of a cost structure and describe a technique for generating an intensional category $\mathcal{C}^{\mathrm{I}}$ plus a cost structure on it from a structure defined solely on $\mathcal{C}^{\mathrm{E}}$.

3.3.1 Semantic definitions with cost

Figure 2.14 in the previous chapter lists a denotational semantics using the lifting monad. The lifting monad handles the strictness properties of the semantics; however, these strictness properties are also closely related to cost properties; if part of an expression is not evaluated, then there is no cost associated with it, and it can be safely replaced with a non-terminating expression. Therefore the first step in converting the extensional semantics to an intensional one is to replace all of the items associated with the lifting monad with the equivalent items from the cost structure. We can safely replace them because a cost structure has lifting built into it: that is why E converts C to L (instead of I) and why CA must be strict.

Specifically, we take the categorical semantics with lifts, and replace all uses of L with C, up with η, $(-)^{\perp}$ with $(-)^*$, and smash with ψ. We also replace the function $\mathsf{raise}_{\perp}$ with raise, where $\mathsf{raise}(f) = \eta \circ \mathsf{curry}(f)$. The change to the cost structure, however, requires a change in categories, so we must find meanings of ground types and constructed data types in $\mathcal{C}^{\mathrm{I}}$. Therefore, we assume that for each ground type g there is an object A_g^{V} in $\mathcal{C}^{\mathrm{I}}$ such that $EA_g^{\mathrm{V}} = A_g$. Similarly, we assume that for each type constructor δ of arity n, there is an n-ary functor F_δ^{V} on $\mathcal{C}^{\mathrm{I}}$ such that for all objects $A_1, \ldots, A_n$ in $\mathcal{C}^{\mathrm{I}}$, $EF_\delta^{\mathrm{V}}(A_1, \ldots, A_n) = F_\delta(EA_1, \ldots EA_n)$. As the meaning of most constants are not specified, we cannot define $\mathcal{C}^{\mathrm{V}}[\![c]\!]$ in terms of its extensional meaning $\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![c]\!]$; instead, for each constant c we assume that $\mathcal{C}^{\mathrm{V}}[\![c]\!]$ is chosen so that $E\mathcal{C}^{\mathrm{V}}[\![c]\!] = \mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![c]\!]$. For conditionals, we specified that $\mathcal{C}^{\mathrm{I}}$ has conditional objects, so $A_{\mathbf{bool}}^{\mathrm{V}}$ is that object, $\mathcal{C}^{\mathrm{V}}[\![\mathtt{true}]\!] = \eta \circ \mathsf{tt}$, and $\mathcal{C}^{\mathrm{V}}[\![\mathtt{false}]\!] = \eta \circ \mathsf{ff}$. Therefore, converting the lifting monad to the cost structure changes the semantics to the one in Figure 3.5.

We still lack an intensional semantics; as currently defined, all expressions would have 0 cost. To add cost explicitly, we must determine the points where costs are added in the operational semantics and add them to the denotational semantics at equivalent points.

There are four operational rules that add extra cost: the two conditional rules, the rule for recursion, and the rule for applying an abstraction. With the morphisms in a cost structure, it is possible to add cost simply by composing part of the semantic definition with $[\![t]\!]$ for some cost t. For the conditionals, each branch has a different cost; therefore, we must add the additional costs to the meanings of e_2 and e_3. For recursion, the additional cost occurs with each expansion; this means that we should add $[\![t_{\mathrm{rec}}]\!]$ within the fixed point constructor. Lastly, for application of the λ-expression, the added cost cannot be added into the denotational definition of application because that definition covers application of constants as well, where there is no added cost. We can, however, add it into the curried part of the definition of an abstraction; the cost then would not appear until the abstraction was applied. The final intensional semantics appears in Figure 3.6.

By design, this semantics is closely related to the extensional semantics, and the separability conditions of the cost structure ensure that we can recover the extensional semantics from the intensional. In particular, for any type τ, $E\mathcal{T}^{\mathrm{V}}[\![\tau]\!] = \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ and whenever $\Gamma \vdash e : \tau$, then $E\mathcal{V}[\![\Gamma \vdash e : \tau]\!] = \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!]$, that is, applying E to the intensional semantics returns the extensional semantics. This is proven formally below:

$$
\begin{array}{rcl}
\mathcal{T}^{\mathrm{V}}[\![g]\!] & = & A^{\mathrm{V}}_g \\
\mathcal{T}^{\mathrm{V}}[\![\delta(\tau_1,\ldots,\tau_n)]\!] & = & F^{\mathrm{V}}_\delta(\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!],\ldots,\mathcal{T}^{\mathrm{V}}[\![\tau_n]\!]) \\
\mathcal{T}^{\mathrm{V}}[\![\tau_1 \to \tau_2]\!] & = & [\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!] \Rightarrow C\mathcal{T}^{\mathrm{V}}[\![\tau_2]\!]] \\
\mathcal{T}^{\mathrm{V}}[\![x_1:\tau_1,\ldots,x_n:\tau_n]\!] & = & \times(C\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!],\ldots,C\mathcal{T}^{\mathrm{V}}[\![\tau_n]\!])
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{V}[\![\Gamma \vdash c:\tau]\!] & = & \mathsf{raise}^{(\mathrm{ar}(c))}(\mathcal{C}^{\mathrm{V}}[\![c:\tau]\!]) \circ !_{\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]} \\
\mathcal{V}[\![\Gamma \vdash x_i:\tau_i]\!] & = & \pi^n_i \\
\mathcal{V}[\![\Gamma \vdash \mathtt{lam}\ x.e:\tau' \to \tau]\!] & = & \eta \circ \mathsf{curry}(\mathcal{V}[\![\Gamma, x:\tau' \vdash e:\tau]\!] \circ (\mathsf{id} \times \eta)) \\
\mathcal{V}[\![\Gamma \vdash e_1(e_2):\tau]\!] & = & (\mathsf{app})^* \circ \psi \circ \langle \mathcal{V}[\![\Gamma \vdash e_1:\tau' \to \tau]\!], \mathcal{V}[\![\Gamma \vdash e_2:\tau']\!]\rangle \\
\mathcal{V}[\![\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3:\tau]\!] & = & (\mathsf{cond})^* \circ \psi \circ \langle \mathcal{V}[\![\Gamma \vdash e_1:\mathbf{bool}]\!], \\
 & & \quad \eta \circ \langle \mathcal{V}[\![\Gamma \vdash e_2:\tau]\!], \mathcal{V}[\![\Gamma \vdash e_3:\tau]\!]\rangle\rangle \\
\mathcal{V}[\![\Gamma \vdash \mathtt{rec}\ x.e:\tau]\!] & = & \mathsf{fixp}(\mathcal{V}[\![\Gamma, x:\tau \vdash e:\tau]\!]) \\
\mathcal{C}^{\mathrm{V}}[\![\mathtt{true}:\mathbf{bool}]\!] & = & \eta \circ \mathsf{tt} \\
\mathcal{C}^{\mathrm{V}}[\![\mathtt{false}:\mathbf{bool}]\!] & = & \eta \circ \mathsf{ff}
\end{array}
$$

where $\mathsf{raise}(f) = \eta \circ \mathsf{curry}(f)$.

Figure 3.5: Call-by-value intensional semantics before cost is added

$$
\begin{array}{rcl}
\mathcal{T}^{\mathrm{V}}[\![g]\!] & = & A^{\mathrm{V}}_g \\
\mathcal{T}^{\mathrm{V}}[\![\delta(\tau_1,\ldots,\tau_n)]\!] & = & F^{\mathrm{V}}_\delta(\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!],\ldots,\mathcal{T}^{\mathrm{V}}[\![\tau_n]\!]) \\
\mathcal{T}^{\mathrm{V}}[\![\tau_1 \to \tau_2]\!] & = & [\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!] \Rightarrow C\mathcal{T}^{\mathrm{V}}[\![\tau_2]\!]] \\
\mathcal{T}^{\mathrm{V}}[\![x_1:\tau_1,\ldots,x_n:\tau_n]\!] & = & \times(C\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!],\ldots,C\mathcal{T}^{\mathrm{V}}[\![\tau_n]\!])
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{V}[\![\Gamma \vdash c:\tau]\!] & = & \mathsf{raise}^{(\mathrm{ar}(c))}(\mathcal{C}^{\mathrm{V}}[\![c:\tau]\!]) \circ !_{\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]} \\
\mathcal{V}[\![\Gamma \vdash x_i:\tau_i]\!] & = & \pi^n_i \\
\mathcal{V}[\![\Gamma \vdash \mathtt{lam}\ x.e:\tau' \to \tau]\!] & = & \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma, x:\tau' \vdash e:\tau]\!] \circ (\mathsf{id} \times \eta)) \\
\mathcal{V}[\![\Gamma \vdash e_1(e_2):\tau]\!] & = & (\mathsf{app})^* \circ \psi \circ \langle \mathcal{V}[\![\Gamma \vdash e_1:\tau' \to \tau]\!], \mathcal{V}[\![\Gamma \vdash e_2:\tau']\!]\rangle \\
\mathcal{V}[\![\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3:\tau]\!] & = & (\mathsf{cond})^* \circ \psi \circ \langle \mathcal{V}[\![\Gamma \vdash e_1:\mathbf{bool}]\!], \\
 & & \quad \eta \circ \langle [\![t_{\mathrm{true}}]\!] \circ \mathcal{V}[\![\Gamma \vdash e_2:\tau]\!], \\
 & & \qquad [\![t_{\mathrm{false}}]\!] \circ \mathcal{V}[\![\Gamma \vdash e_3:\tau]\!]\rangle\rangle \\
\mathcal{V}[\![\Gamma \vdash \mathtt{rec}\ x.e:\tau]\!] & = & \mathsf{fixp}([\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, x:\tau \vdash e:\tau]\!]) \\
\mathcal{C}^{\mathrm{V}}[\![\mathtt{true}:\mathbf{bool}]\!] & = & \eta \circ \mathsf{tt} \\
\mathcal{C}^{\mathrm{V}}[\![\mathtt{false}:\mathbf{bool}]\!] & = & \eta \circ \mathsf{ff}
\end{array}
$$

where $\mathsf{raise}(f) = \eta \circ \mathsf{curry}(f)$.

Figure 3.6: Call-by-value intensional semantics

Lemma 3.3.2 *For all n, $E\mathsf{raise}^{(n)}(f) = \mathsf{raise}_{\perp}^{(n)}(Ef)$.*

Proof. By straightforward induction on n. ◻

Theorem 3.3.3 *For all types τ, $E\mathcal{T}^{\mathrm{V}}[\![\tau]\!] = \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$.*

Proof. By induction on the structure of τ. All cases are straightforward, we show the case for function types.

Case $\tau = \tau_1 \to \tau_2$:

$$\begin{array}{rll}
E\mathcal{T}^{\mathrm{V}}[\![\tau_1 \to \tau_2]\!] = & E[\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!] \Rightarrow C\mathcal{T}^{\mathrm{V}}[\![\tau_2]\!]] & \\
= & [E\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!] \Rightarrow EC\mathcal{T}^{\mathrm{V}}[\![\tau_2]\!]] & \\
= & [E\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!] \Rightarrow LE\mathcal{T}^{\mathrm{V}}[\![\tau_2]\!]] & \text{separability of } E \\
= & [\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1]\!] \Rightarrow L\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_2]\!]] & \text{induction hypothesis} \\
= & \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1 \to \tau_2]\!] &
\end{array}$$

◻

Lemma 3.3.4 *For all type environments Γ, $E\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!] = \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\Gamma]\!]$.*

Proof. By straightforward induction on the size of Γ. ◻

Theorem 3.3.5 *For any expressions e such that $\Gamma \vdash e : \tau$, $E\mathcal{V}[\![\Gamma \vdash e : \tau]\!] = \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!]$.*

Proof. By induction on the structure of e. Again the cases are straightforward; we include the case for conditionals.

Case $e = \mathcal{V}[\![\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau]\!]$:

$$\begin{array}{ll}
\multicolumn{2}{l}{E\mathcal{V}[\![\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau]\!]} \\
= & E(\mathsf{cond}^* \circ \psi \circ \langle \mathcal{V}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!], \\
& \quad \eta \circ \langle [\![t_{\mathrm{true}}]\!] \circ \mathcal{V}[\![\Gamma \vdash e_2 : \tau]\!], [\![t_{\mathrm{false}}]\!] \circ \mathcal{V}[\![\Gamma \vdash e_3 : \tau]\!]\rangle\rangle \\
= & E\mathsf{cond}^* \circ E\psi \circ \langle E\mathcal{V}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!], \\
& \quad E\eta \circ \langle E[\![t_{\mathrm{true}}]\!] \circ E\mathcal{V}[\![\Gamma \vdash e_2 : \tau]\!], E[\![t_{\mathrm{false}}]\!] \circ E\mathcal{V}[\![\Gamma \vdash e_3 : \tau]\!]\rangle\rangle) \\
& \hfill E \text{ is a representation} \\
= & (E\mathsf{cond})^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!], \\
& \quad \mathsf{up} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_3 : \tau]\!]\rangle\rangle \\
& \hfill \text{separability of } E \text{ and induction hypothesis} \\
= & \mathsf{cond}^{\perp} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!], \\
& \quad \mathsf{up} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_3 : \tau]\!]\rangle\rangle \\
= & \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau]\!]
\end{array}$$

◻

We will use this relationship to prove both that the extensional semantics is sound (based on the soundness of the intensional semantics) and that the intensional semantics is adequate (based on the adequacy of the extensional semantics).

3.4 Soundness of the categorical model

The extensional semantics is sound if whenever $e \Rightarrow_{\mathrm{v}} v$ and e has type τ, then $\mathcal{V}_{\mathrm{E}}[\![\vdash e : \tau]\!]$ equals $\mathcal{V}_{\mathrm{E}}[\![\vdash v : \tau]\!]$. For the intensional semantics, we would not expect the meaning of e to be the same as v because v has no cost associated with its evaluation while e does. Adding the cost t to the meaning of v results in the equation $\mathcal{V}[\![\vdash e : \tau]\!] = [\![t]\!] \circ \mathcal{V}[\![\vdash v : \tau]\!]$. In this section we prove that this equation holds for the intensional semantics when $e \overset{t}{\Rightarrow}_{\mathrm{v}} v$.

The soundness proof follows the same general outline as the soundness proof in [11], with a few notable exceptions. One difference, of course, is the addition of cost. Another is the need to prove explicitly that values have 0 cost. Lastly, because we handle constants differently than standard sources do, we include special lemmas concerning them.

The soundness proof is divided into five parts. The first part contains general technical lemmas which allow certain simplifications. The second part contains the proof of the substitution lemma and its sub-lemmas. The third part defines the assumptions made about constants and proves some useful lemmas concerning them. The fourth part contains the proof of some critical properties concerning values. Finally the last part gives the soundness proof.

3.4.1 Technical Lemmas

Properties of n-ary products and C

Definition 3.4.1 Given a category $\mathcal{C}$ with finite products and objects $A_1, \ldots, A_n$, for $1 \le i \le n$, let β_i^n be the natural isomorphism from $\times(A_1, \ldots, A_{i-1}, A_i, \ldots, A_n)$ to $\times(A_1, \ldots, A_i, A_{i-1}, \ldots, A_n)$ defined inductively on $n - i$ as follows:

- $\beta_n^n = \langle \pi_1 \times \mathsf{id}, \pi_2 \circ \pi_1 \rangle$
- For $i < n$, $\beta_i^n = \beta_i^{n-1} \times \mathsf{id}_{A_n}$.

Lemma 3.4.1 (β_i^n properties) *If $1 \le i \le n$ and i is not equal to k or $k-1$, the following holds:*

- $\pi_k^n = \pi_{k-1}^n \circ \beta_k^n$
- $\pi_{k-1}^n = \pi_k^n \circ \beta_k^n$
- $\pi_i^n = \pi_i^n \circ \beta_k^n$

where π_k^n is the k'th projection of an n-ary product.

Proof. Clear from the universal properties of products; also see [11], pg. 71. □

Lemma 3.4.2 ($\times(-)$ property) *Let $A_1, \ldots, A_n, C, C'$ be objects in $\mathcal{C}$ and for $1 \le i \le n$, let $f_i : C \to A_i$ and $g : C' \to C$. Then*

$$\langle\rangle(f_1, \ldots, f_n) \circ g = \langle\rangle(f_1 \circ g, \ldots, f_n \circ g)$$

Proof. Clear from the universal properties of products. □

The next lemma covers simplifications used frequently in the soundness proof.

Lemma 3.4.3 *For any $h : A \times B \to CD$, $f : D' \to A$, $g : D' \to B$, and $t_1, t_2 \in T$,*

1. $h^* \circ \psi_{A,B} \circ \langle [\![t_1]\!]_A \circ \eta_A \circ f, [\![t_2]\!]_B \circ \eta_B \circ g\rangle = [\![t_2 + t_1]\!]_D \circ h \circ \langle f, g\rangle$

2. $\mathsf{app}^* \circ \psi_{A,B} \circ \langle [\![t_1]\!]_A \circ \eta_A \circ \mathsf{curry}(h) \circ f, [\![t_2]\!]_B \circ \eta_B \circ g\rangle = [\![t_2 + t_1]\!]_D \circ h \circ \langle f, g\rangle$

Proof. Follows easily from the properties of products and cost structures. □

By setting t_1 or t_2 to 0 we also have lemmas for the equations that lack added costs.

Switch, Drop and Substitution lemmas

With a precise definition for n-ary products, the order and number of variables in a type environment affect its meaning (though the different meanings are equivalent). We need the Switch and Drop lemmas to describe the differences, making it possible to prove the Substitution Lemma.

Lemma 3.4.4 (Switch Lemma) *Let*

$$\Gamma = x_1 : \tau_1, \ldots, x_{k-1} : \tau_{k-1}, x_k : \tau_k, \ldots, x_n : \tau_n$$

$$\Gamma' = x_1 : \tau_1, \ldots, x_k : \tau_k, x_{k-1} : \tau_{k-1}, \ldots, x_n : \tau_n$$

Then if $\Gamma \vdash e : \tau$,

$$\mathcal{V}[\![\Gamma \vdash e : \tau]\!] = \mathcal{V}[\![\Gamma' \vdash e : \tau]\!] \circ \beta^n_k$$

Proof. By straightforward induction on the structure of e. □

Lemma 3.4.5 (Drop Lemma) *If* $\Gamma \vdash e : \tau$ *and* x *is not free in* e*, then*

$$\mathcal{V}[\![\Gamma, x : \tau' \vdash e : \tau]\!] = \mathcal{V}[\![\Gamma \vdash e : \tau]\!] \circ \pi_1$$

Proof. By straightforward induction on the structure of e. □

Lemma 3.4.6 (Substitution Lemma) *Suppose that* $\Gamma, x : \tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$*. Then*

$$\mathcal{V}[\![\Gamma \vdash [e'/x]e : \tau]\!] = \mathcal{V}[\![\Gamma, x : \tau' \vdash e : \tau]\!] \circ \langle \mathsf{id}, \mathcal{V}[\![\Gamma \vdash e' : \tau']\!]\rangle$$

Proof. By induction on the structure of e. All cases are straightforward; we list one of the variable cases and the case when e is an abstraction below.

Case $e = x$ $(\tau' = \tau)$**:**

$$\begin{aligned}\mathcal{V}[\![\Gamma \vdash [e'/x]x : \tau]\!] &= \mathcal{V}[\![\Gamma \vdash e' : \tau]\!] \\ &= \pi_2 \circ \langle \mathsf{id}, \mathcal{V}[\![\Gamma \vdash e' : \tau]\!]\rangle \\ &= \mathcal{V}[\![\Gamma, x : \tau \vdash x : \tau]\!] \circ \langle \mathsf{id}, \mathcal{V}[\![\Gamma \vdash e' : \tau]\!]\rangle\end{aligned}$$

$$\begin{tikzcd}
\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!] \times \mathcal{T}^{\mathrm{V}}[\![\tau_1]\!] \arrow[r, "\langle \mathsf{id}, g\rangle \times \mathsf{id}"] \arrow[d, "\mathsf{id} \times \eta"'] & (\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!] \times C\mathcal{T}^{\mathrm{V}}[\![\tau']\!]) \times \mathcal{T}^{\mathrm{V}}[\![\tau_1]\!] \arrow[d, "\mathsf{id} \times \eta"] \\
\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!] \times C\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!] \arrow[r, "\langle \mathsf{id}, g\rangle \times \mathsf{id}"] \arrow[d, "\langle \mathsf{id}, g \circ \pi_1\rangle"'] & (\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!] \times C\mathcal{T}^{\mathrm{V}}[\![\tau']\!]) \times C\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!] \arrow[d, "\mathcal{V}[\![\Gamma, x : \tau', y : \tau_1 \vdash e'' : \tau_2]\!]"] \\
(\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!] \times C\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!]) \times C\mathcal{T}^{\mathrm{V}}[\![\tau']\!] \arrow[ur, "\beta^{(k+2)}_{(k+2)}"] \arrow[r, "\mathcal{V}[\![\Gamma, y : \tau_1, x : \tau' \vdash e'' : \tau_2]\!]"'] & C\mathcal{T}^{\mathrm{V}}[\![\tau_2]\!]
\end{tikzcd}$$

Figure 3.7: Part of the proof of Substitution Lemma, when $e = \texttt{lam}\ y.e''$. The lower right triangle follows from the Switch Lemma

Case $e = \texttt{lam}\ y.e''$ $(\tau = \tau_1 \to \tau_2)$**:**

Let $g = \mathcal{V}[\![\Gamma \vdash e' : \tau']\!]$ and n be the number of variables in Γ. We can assume that $y \neq x$ and y is not free in e'. By the induction hypothesis

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{V}[\![\Gamma \vdash [e'/x]\texttt{lam}\ y.e'' : \tau_1 \to \tau_2]\!]} \\ = & \mathcal{V}[\![\Gamma \vdash \texttt{lam}\ y.[e'/x]e'' : \tau_1 \to \tau_2]\!] \\ = & \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma, y : \tau_1 \vdash [e'/x]e'' : \tau_2]\!] \circ (\mathsf{id} \times \eta)) \\ = & \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma, y : \tau_1, x : \tau' \vdash e'' : \tau_2]\!] \\ & \quad \circ \langle \mathsf{id}, \mathcal{V}[\![\Gamma, y : \tau_1 \vdash e' : \tau']\!]\rangle \circ (\mathsf{id} \times \eta)) \end{array}$$

By the Drop Lemma $\mathcal{V}[\![\Gamma, y : \tau_1 \vdash e' : \tau']\!] = g \circ \pi_2$. Figure 3.7 shows that

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{V}[\![\Gamma, y : \tau_1, x : \tau' \vdash e'' : \tau_2]\!] \circ \langle \mathsf{id}, g \circ \pi_2\rangle \circ (\mathsf{id} \times \eta)} \\ = & \mathcal{V}[\![\Gamma, x : \tau', y : \tau_2 \vdash e'' : \tau_2]\!] \circ (\mathsf{id} \times \eta) \circ (\langle \mathsf{id}, g\rangle \times \mathsf{id}) \end{array}$$

Thus

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{V}[\![\Gamma \vdash [e'/x]\texttt{lam}\ y.e'' : \tau_1 \to \tau_2]\!]} \\ = & \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau', y : \tau_1 \vdash e'' : \tau_2]\!] \circ (\mathsf{id} \times \eta) \circ (\langle \mathsf{id}, g\rangle \times \mathsf{id})) \\ = & \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau', y : \tau_1 \vdash e'' : \tau_2]\!] \circ (\mathsf{id} \times \eta)) \circ \langle \mathsf{id}, g\rangle \\ = & \mathcal{V}[\![\Gamma, x : \tau' \vdash \texttt{lam}\ y.e'' : \tau_1 \to \tau_2]\!] \circ \langle \mathsf{id}, g\rangle \end{array}$$

as required.

□

3.4.2 Constants

Even though we do not know precisely what constants may be in use, we must make some assumptions about the meaning of constants in order to prove soundness. These assumptions are in two forms: simple structural assumptions about constructors (whose behaviors are known) and more complicated assumptions about fully applied non-constructor constants. We do not have to assume anything about partially applied constants; we already know their behavior, and the structure given them through the raise function has sufficient information for the soundness proof.

The assumptions made for non-constructor constants are more complicated than those for constructors because their operational rules in some cases require additional evaluation. For example, one of the operational rules for the `case` constant is

$$\frac{v_1(v) \overset{t}{\Rightarrow}_{\mathrm{v}} v'}{\mathrm{apply}(\texttt{case inl}(v)\ v_1, v_2) \overset{t+t_{\mathrm{case}}}{\Rightarrow} v'}$$

To prove that a particular meaning for `case` is sound, we need to be able to assume a soundness relation between $v_1(v)$ and v'. Given that the soundness proof depends on the soundness of constants, one cannot safely assume that a soundness relationship exists. The relation, however, will hold (via a structural induction hypothesis) whenever we need the soundness of `case`. Therefore it will be sufficient to define soundness of constants relative to specific soundness assumptions about sub-evaluations.

Definition 3.4.2 $\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!]$ is *sound* at c if the following two properties hold:

1. If c is a constructor then $\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!]$ factors through $\eta_{\mathcal{T}^{\mathrm{V}}[\![\tau]\!]}$, that is, there exists a morphism g such that $\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!] = \eta_{\mathcal{T}^{\mathrm{V}}[\![\tau]\!]} \circ g$. Otherwise

2. For all operational rules of the form

$$\frac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{v}} v'_1 \ldots e_k \overset{t_k}{\Rightarrow}_{\mathrm{v}} v'_k}{\mathrm{vapply}(c, v_1, \ldots, v_n) \overset{t}{\Rightarrow} v}$$

   where n is the arity of c, suppose that for all $1 \le i \le k$ and any types τ'_i, $\vdash e_i : \tau'_i$ implies that $\mathcal{V}[\![e_i : \tau'_i]\!] = [\![t_i]\!] \circ \mathcal{V}[\![v'_i : \tau'_i]\!]$. Then if c has type $\tau_1 \to \ldots \to \tau_n \to \tau'$ and for $1 \le j \le n$, each value v_j has type τ_j and there exist a morphism f_j such that $\mathcal{V}[\![v_j : \tau_j]\!] = \eta_{\mathcal{T}^{\mathrm{V}}[\![\tau_j]\!]} \circ f_j$,

$$\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!] \circ \langle\rangle(f_1, \ldots, f_n) = [\![t]\!] \circ \mathcal{V}[\![v : \tau']\!]$$

$\mathcal{C}^{\mathrm{V}}[\![-]\!]$ is *sound* if for all $c \in \mathbf{Const}_\tau$, $\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!]$ is sound at c.

Note that $\mathcal{C}^{\mathrm{V}}[\![\texttt{true} : \mathbf{bool}]\!]$ is sound at `true` and $\mathcal{C}^{\mathrm{V}}[\![\texttt{false} : \mathbf{bool}]\!]$ is sound at `false` because they are both constructors (of arity 0) and their meanings factor through η.

Applied constants

One reason we defined $\mathcal{C}^{\mathrm{V}}[\![-]\!]$ on products and then used raise to convert the meaning of constants is the desire to simplify the meaning of curried constants applied to values. Thus there is a useful lemma describing the meaning of $cv_1 \ldots v_n$:

Lemma 3.4.7 (Constant Application) *Suppose that* $c \in \mathbf{Const}_{\tau_1 \to \ldots \to \tau_n \to \tau}$ *has arity* n*, that* $0 \le i \le n$*, and that for* $1 \le j \le i$*,* $\Gamma \vdash v_j : \tau_j$*. Furthermore suppose that for* $1 \le j \le i$*, there exists a morphism* f_j *such that*

$$\mathcal{V}[\![\Gamma \vdash v_j : \tau]\!] = \eta \circ f_j$$

Then

$$\mathcal{V}[\![\Gamma \vdash cv_1 \ldots v_i : \tau_{i+1} \to \ldots \to \tau_n \to \tau]\!] = \mathsf{raise}^{(n-i)}(\mathcal{C}^{\mathrm{V}}[\![c]\!]) \circ \langle\rangle(f_1, \ldots, f_i)$$

Proof. By induction on i. Let $\tau' = \tau_{i+1} \to \ldots \to \tau_n \to \tau$. If $i = 0$ then

$$\begin{aligned}\mathcal{V}[\![\Gamma \vdash c : \tau]\!] &= \mathsf{raise}^{(n)}(\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!]) \circ !_{\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]} \\ &= \mathsf{raise}^{(n-0)}(\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!]) \circ \langle\rangle()\end{aligned}$$

If $i > 0$ then

$$\begin{aligned}&\mathcal{V}[\![\Gamma \vdash cv_1 \ldots v_i : \tau']\!] \\ &= \mathcal{V}[\![\Gamma \vdash (cv_1 \ldots v_{i-1})(v_i) : \tau']\!] \\ &= \mathsf{app}^* \circ \psi \circ \langle \mathcal{V}[\![\Gamma \vdash cv_1 \ldots v_{i-1} : \tau_i \to \tau']\!], \mathcal{V}[\![\Gamma \vdash v_i : \tau_i]\!]\rangle \\ &= \mathsf{app}^* \circ \psi \circ \langle \mathsf{raise}^{(n-(i-1))}(\mathcal{C}^{\mathrm{V}}[\![c]\!]) \circ \langle\rangle(f_1, \ldots, f_{i-1}), \qquad \text{induction hypothesis} \\ &\qquad \mathcal{V}[\![\Gamma \vdash e_i : \tau_i]\!]\rangle \\ &= \mathsf{app}^* \circ \psi \circ \langle \mathsf{raise}^{((n-i)+1)}(\mathcal{C}^{\mathrm{V}}[\![c]\!]) \circ \langle\rangle(f_1, \ldots, f_{i-1}), \eta \circ f_i\rangle \quad \text{by assumption} \\ &= \mathsf{app}^* \circ \psi \circ \langle \eta \circ \mathsf{curry}(\mathsf{raise}^{(n-i)}(\mathcal{C}^{\mathrm{V}}[\![c]\!])) \circ \langle\rangle(f_1, \ldots, f_{i-1}), \eta \circ f_i\rangle \\ &= \mathsf{raise}^{(n-i)}(\mathcal{C}^{\mathrm{V}}[\![c]\!]) \circ \langle\langle\rangle(f_1, \ldots, f_{i-1}), f_i\rangle \qquad \text{Lemma 3.4.3} \\ &= \mathsf{raise}^{(n-i)}(\mathcal{C}^{\mathrm{V}}[\![c]\!]) \circ \langle\rangle(f_1, \ldots, f_i)\end{aligned}$$

□

3.4.3 Values

The next step shows that values have zero cost denotationally. Because we need this lemma available to prove specific constants sound, we cannot assume in its premise that all constants are sound. We can, however, safely assume that constructors are sound.

Lemma 3.4.8 *Suppose that for all constructors c, $\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!]$ factors through η. Then for any value v and type enviroment Γ such that $\Gamma \vdash v : \tau$, $\mathcal{V}[\![\Gamma \vdash v : \tau]\!]$ factors through η.*

Proof. By induction on the structure of v (as a value).

Case $v = c$:

If the arity of c is 0, then by assumption there exists a morphism f' such that $\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!] = \eta \circ f'$. Thus

$$\begin{aligned}\mathcal{V}[\![\Gamma \vdash c : \tau]\!] &= \mathsf{raise}^{(0)}(\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!]) \circ !_{\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]} \\ &= \mathcal{C}^{\mathrm{V}}[\![c : \tau]\!] \circ !_{\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]} \\ &= \eta \circ f' \circ !_{\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]}\end{aligned}$$

If the arity of c is not 0, then

$$\begin{aligned}\mathcal{V}[\![\Gamma \vdash c : \tau]\!] &= \mathsf{raise}^{(\mathrm{ar}(c))}(\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!]) \circ !_{\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]} \\ &= \eta \circ \mathsf{curry}(\mathsf{raise}^{(\mathrm{ar}(c)-1)}(\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!])) \circ !_{\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]}\end{aligned}$$

Case $v = \mathtt{lam}\ x.e$: Follows directly from the definition of $\mathcal{V}[\![\Gamma \vdash \mathtt{lam}\ x.e : \tau' \to \tau]\!]$.

Case $v = cv_1 \ldots v_i$, $i < \mathrm{ar}(c)$:

Because $\Gamma \vdash cv_1 \ldots v_i : \tau$, there exist types $\tau_1, \ldots, \tau_n$, where n is the arity of c, and a type τ' such that $c \in \mathbf{Const}_{\tau_1 \to \ldots \to \tau_n \to \tau'}$, $\tau = \tau_{i+1} \to \ldots \to \tau_n \to \tau'$ and for each $1 \leq j \leq i$,

$\Gamma \vdash v_j : \tau_j$. Also as each v_j is a value, by the induction hypothesis there exists an f_j such that $\mathcal{V}[\![\Gamma \vdash v_j : \tau_j]\!] = \eta \circ f_j$. Thus

$$\begin{array}{lll} \mathcal{V}[\![\Gamma \vdash cv_1 \ldots v_i : \tau]\!] & & \\ \quad = & \mathsf{raise}^{(n-i)}(\mathcal{C}^{\mathrm{V}}[\![c]\!]) \circ \langle\rangle(f_1, \ldots, f_i) & \text{Lemma 3.4.7} \\ \quad = & \eta \circ \mathsf{curry}(\mathsf{raise}^{(n-i-1)}(\mathcal{C}^{\mathrm{V}}[\![c]\!])) \circ \langle\rangle(f_1, \ldots, f_i) & i < n \end{array}$$

Case $v = cv_1 \ldots v_{\mathrm{ar}(c)}$, $c \in$ Construct:

Let $n = \mathrm{ar}(c)$. As $\Gamma \vdash v : \tau$, there exist types $\tau_1, \ldots, \tau_n$ such that $c \in \mathbf{Construct}_{\tau_1 \to \ldots \to \tau_n \to \tau}$ and for each $1 \leq i \leq n$, $\Gamma \vdash v_i : \tau_i$. Also as each v_i is a value, by the induction hypothesis there exists an f_i such that $\mathcal{V}[\![\Gamma \vdash v_i : \tau_i]\!] = \eta \circ f_i$. Furthermore as c is a constructor, there exists a morphismf' such that $\mathcal{C}^{\mathrm{V}}[\![c]\!] = \eta \circ f'$. Therefore

$$\begin{array}{lll} \mathcal{V}[\![\Gamma \vdash cv_1 \ldots v_n : \tau]\!] & & \\ \quad = & \mathsf{raise}^{(n-n)}(\mathcal{C}^{\mathrm{V}}[\![c]\!]) \circ \langle\rangle(f_1, \ldots, f_n) & \text{Lemma 3.4.7} \\ \quad = & \mathcal{C}^{\mathrm{V}}[\![c]\!] \circ \langle\rangle(f_1, \ldots, f_n) & \\ \quad = & \eta \circ f' \circ \langle\rangle(f_1, \ldots, f_n) & \end{array}$$

□

3.4.4 Soundness proof

Theorem 3.4.9 *Suppose $\vdash e : \tau$ and $e \overset{t}{\Rightarrow}_{\mathrm{v}} v$ and $\mathcal{C}^{\mathrm{V}}[\![c]\!]$ is sound for each constant c. Then*

$$\mathcal{V}[\![e : \tau]\!] = [\![t]\!] \circ \mathcal{V}[\![v : \tau]\!]$$

Proof. By induction on the structure of the derivation of $e \overset{t}{\Rightarrow}_{\mathrm{v}} v$.

Case $c \overset{0}{\Rightarrow}_{\mathrm{v}} c$ and $\mathtt{lam}\ x.e \overset{0}{\Rightarrow}_{\mathrm{v}} \mathtt{lam}\ x.e$:

Both follow directly from the soundness property stating that $[\![0]\!] = \mathsf{id}$.

Case $\dfrac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{v}} \mathtt{lam}\ x.e' \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{v}} v' \quad [v'/x]e' \overset{t'}{\Rightarrow}_{\mathrm{v}} v}{e_1(e_2) \overset{t'+t_{\mathrm{app}}+t_2+t_1}{\Rightarrow}_{\mathrm{v}} v}$**:**

Because $\vdash e_1(e_2) : \tau$, there exists a type τ' such that $\vdash e_1 : \tau' \to \tau$ and $\vdash e_2 : \tau'$.

By value soundness, we know that v' is a value, so by Lemma 3.4.8 there exists a morphism $y' : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau']\!]$ such that $\mathcal{V}[\![v' : \tau']\!] = \eta \circ y'$. Furthermore, by the induction hypothesis

$$\begin{array}{rcl} \mathcal{V}[\![e_1 : \tau' \to \tau]\!] & = & [\![t_1]\!] \circ \mathcal{V}[\![\mathtt{lam}\ x.e' : \tau' \to \tau]\!] \\ & = & [\![t_1]\!] \circ \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![x : \tau' \vdash e' : \tau]\!] \circ (\mathsf{id} \times \eta)), \\ \mathcal{V}[\![e_2 : \tau']\!] & = & [\![t_2]\!] \circ \mathcal{V}[\![v' : \tau']\!] = [\![t_2]\!] \circ \eta \circ y', \text{ and} \\ \mathcal{V}[\![[e_2/x]e' : \tau]\!] & = & [\![t']\!] \circ \mathcal{V}[\![v : \tau]\!] \end{array}$$

Thus

$$
\begin{array}{rll}
\mathcal{V}[\![e_1(e_2) : \tau]\!] & & \\
= & \mathsf{app}^* \circ \psi \circ \langle \mathcal{V}[\![e_1 : \tau' \to \tau]\!], \mathcal{V}[\![e_2 : \tau']\!]\rangle & \\
= & \mathsf{app}^* \circ \psi \circ \langle [\![t_1]\!] \circ \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![x : \tau' \vdash e' : \tau]\!] & \\
 & \quad \circ (\mathsf{id} \times \eta)), [\![t_2]\!] \circ \eta \circ y'\rangle & \\
= & [\![t_2 + t_1]\!] \circ [\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![x : \tau' \vdash e' : \tau]\!] \circ (\mathsf{id} \times \eta) \circ \langle \mathsf{id}, y'\rangle & \text{Lemma 3.4.3} \\
= & [\![t_2 + t_1]\!] \circ [\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![x : \tau' \vdash e' : \tau]\!] \circ \langle \mathsf{id}, \mathcal{V}[\![v' : \tau']\!]\rangle & \\
= & [\![t_2 + t_1]\!] \circ [\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![[v'/x]e' : \tau]\!] & \text{Substitution Lemma} \\
= & [\![t_2 + t_1]\!] \circ [\![t_{\mathrm{app}}]\!] \circ [\![t']\!] \circ \mathcal{V}[\![v : \tau]\!] & \\
= & [\![t' + t_{\mathrm{app}} + t_2 + t_1]\!] \circ \mathcal{V}[\![v : \tau]\!] &
\end{array}
$$

Case $\dfrac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{v}} cv_1 \ldots v_i \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{v}} v' \quad \mathrm{vapply}(cv_1 \ldots v_i, v') \overset{t'}{\Rightarrow} v}{e_1(e_2) \overset{t'+t_2+t_1}{\Rightarrow}_{\mathrm{v}} v}$**:**

Because $\vdash e_1(e_2) : \tau$, there exists a type τ' such that $\vdash e_1 : \tau' \to \tau$ and $\vdash e_2 : \tau'$. By type soundness this means that $\vdash cv_1 \ldots v_i : \tau' \to \tau$ so there exist types $\tau_1, \ldots, \tau_i$ such that for each $1 \leq j \leq i$, $\vdash v_j : \tau_j$.

By the induction hypothesis

$$\mathcal{V}[\![e_1 : \tau' \to \tau]\!] = [\![t_1]\!] \circ \mathcal{V}[\![cv_1 \ldots v_i : \tau' \to \tau]\!]$$

and

$$\mathcal{V}[\![e_2 : \tau']\!] = [\![t_2]\!] \circ \mathcal{V}[\![v' : \tau']\!]$$

Furthermore, by value soundness each of $v_1, \ldots v_i$ are values, so by Lemma 3.4.8 for each $1 \leq j \leq i$, there exists a morphism y_j such that $\mathcal{V}[\![v_j : \tau_j]\!] = \eta \circ y_j$. Similarly, as v' is a value there exists a morphism y' such that $\mathcal{V}[\![v' : \tau']\!] = \eta \circ y'$. Lastly as $cv_1 \ldots v_i$ is not fully applied, it is a value, thus there exists a morphism y such that $\mathcal{V}[\![cv_1 \ldots v_i : \tau' \to \tau]\!] = \eta \circ y$.

Next note that

$$
\begin{array}{rll}
\mathcal{V}[\![e_1(e_2) : \tau]\!] & & \\
= & \mathsf{app}^* \circ \psi \circ \langle \mathcal{V}[\![e_1 : \tau' \to \tau]\!], \mathcal{V}[\![e_2 : \tau']\!]\rangle & \\
= & \mathsf{app}^* \circ \psi \circ \langle [\![t_1]\!] \circ \mathcal{V}[\![cv_1 \ldots v_i : \tau' \to \tau]\!], [\![t_2]\!] \circ \mathcal{V}[\![v' : \tau']\!]\rangle & \text{induction hypothesis} \\
= & \mathsf{app}^* \circ \psi \circ \langle [\![t_1]\!] \circ \eta \circ y, [\![t_2]\!] \circ \eta \circ y'\rangle & \text{as noted} \\
= & [\![t_2 + t_1]\!] \circ \mathsf{app} \circ \langle y, y'\rangle & \text{Lemma 3.4.3} \\
= & [\![t_2 + t_1]\!] \circ \mathsf{app}^* \circ \psi \circ \langle \eta \circ y, \eta \circ y'\rangle & \text{Lemma 3.4.3} \\
= & [\![t_2 + t_1]\!] \circ \mathsf{app}^* \circ \psi \circ \langle \mathcal{V}[\![cv_1 \ldots v_i : \tau' \to \tau]\!], \mathcal{V}[\![v' : \tau']\!]\rangle & \\
= & [\![t_2 + t_1]\!] \circ \mathcal{V}[\![cv_1 \ldots v_i v' : \tau]\!] &
\end{array}
$$

There are three possible types of derivations for $\mathrm{apply}(cv_1 \ldots v_i, v') \overset{t'}{\Rightarrow} v$: when $i < n-1$, when $i = n-1$ and c is a constructor, or when $i = n-1$ and c is not a constructor. For the first two cases, $v = cv_1 \ldots v_i v'$ and $t' = 0$, so

$$
\begin{aligned}
\mathcal{V}[\![e_1(e_2) : \tau]\!] &= [\![t_2 + t_1]\!] \circ \mathcal{V}[\![cv_1 \ldots v_i v' : \tau]\!] \\
&= [\![t' + t_2 + t_1]\!] \circ \mathcal{V}[\![v : \tau]\!]
\end{aligned}
$$

Otherwise the derivation for $\mathrm{vapply}(cv_1 \ldots v_i, v') \stackrel{t'}{\Rightarrow} v$ is of the form

$$\frac{e'_1 \stackrel{t'_1}{\Rightarrow}_{\mathrm{v}} v'_1 \ldots e'_k \stackrel{t'_k}{\Rightarrow}_{\mathrm{v}} v'_k}{\mathrm{vapply}(cv_1 \ldots v_i, v') \stackrel{t'}{\Rightarrow} v}$$

By the induction hypothesis for any $1 \leq j \leq k$ and any type τ'_j, if e'_1 has type τ'_j then

$$\mathcal{V}[\![e'_j : \tau'_j]\!] = [\![t'_j]\!] \circ \mathcal{V}[\![v'_j : \tau'_j]\!]$$

Therefore by the assumption of soundness for constants

$$\begin{array}{lll} [\![t']\!] \circ \mathcal{V}[\![v : \tau]\!] & & \\ \quad = & \mathcal{C}^{\mathrm{V}}[\![c]\!] \circ \langle\rangle(y_1, \ldots, y_i, y') & \\ \quad = & \mathsf{raise}^{(n-n)}(\mathcal{C}^{\mathrm{V}}[\![c]\!]) \circ \langle\rangle(y_1, \ldots, y_i, y') & \\ \quad = & \mathcal{V}[\![cv_1 \ldots v_i v' : \tau]\!] & \text{Lemma 3.4.7} \\ \quad = & \mathcal{V}[\![v_1(v_2) : \tau]\!] & \end{array}$$

Thus

$$\begin{array}{rcl} \mathcal{V}[\![e_1(e_2) : \tau]\!] & = & [\![t_2 + t_1]\!] \circ \mathcal{V}[\![cv_1 \ldots v_i v' : \tau]\!] \\ & = & [\![t_2 + t_1]\!] \circ [\![t']\!] \circ \mathcal{V}[\![v : \tau]\!] \\ & = & [\![t' + t_2 + t_1]\!] \circ \mathcal{V}[\![v : \tau]\!] \end{array}$$

Case $\dfrac{e_1 \stackrel{t_1}{\Rightarrow}_{\mathrm{v}} \texttt{true} \quad e_2 \stackrel{t_2}{\Rightarrow}_{\mathrm{v}} v}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \stackrel{t_2+t_{\mathrm{true}}+t_1}{\Rightarrow}_{\mathrm{v}} v}$**:**

Because $\vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau$, we know that $\vdash e_1 : \mathbf{bool}$ and $\vdash e_2 : \tau$. Furthermore by the induction hypothesis and Lemma 3.4.8

$$\begin{array}{rcl} \mathcal{V}[\![e_1 : \mathbf{bool}]\!] & = & [\![t_1]\!] \circ \mathcal{V}[\![\texttt{true} : \mathbf{bool}]\!] \\ & = & [\![t_1]\!] \circ \mathcal{C}^{\mathrm{V}}[\![\texttt{true} : \mathbf{bool}]\!] \circ ! \\ & = & [\![t_1]\!] \circ \eta \circ \mathsf{tt} \circ ! \\ & & \text{and} \\ \mathcal{V}[\![e_2 : \tau]\!] & = & [\![t_2]\!] \circ \mathcal{V}[\![v : \tau]\!] \end{array}$$

Therefore

$$\begin{array}{lll} \mathcal{V}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau]\!] & & \\ \quad = & \mathsf{cond}^* \circ \psi \circ \langle \mathcal{V}[\![e_1 : \mathbf{bool}]\!], \eta \circ \langle [\![t_{\mathrm{true}}]\!] \circ \mathcal{V}[\![e_2 : \tau]\!], & \\ & \quad [\![t_{\mathrm{false}}]\!] \circ \mathcal{V}[\![e_3 : \tau]\!]\rangle\rangle & \\ \quad = & \mathsf{cond}^* \circ \psi \circ \langle [\![t_1]\!] \circ \eta \circ \mathsf{tt} \circ !, \eta \circ \langle [\![t_{\mathrm{true}}]\!] \circ \mathcal{V}[\![e_2 : \tau]\!], & \text{as noted} \\ & \quad [\![t_{\mathrm{false}}]\!] \circ \mathcal{V}[\![e_3 : \tau]\!]\rangle\rangle & \\ \quad = & [\![t_1]\!] \circ \mathsf{cond} \circ \langle \mathsf{tt} \circ !, \langle [\![t_{\mathrm{true}}]\!] \circ \mathcal{V}[\![e_2 : \tau]\!], [\![t_{\mathrm{false}}]\!] \circ \mathcal{V}[\![e_3 : \tau]\!]\rangle\rangle & \text{Lemma 3.4.3} \\ \quad = & [\![t_1]\!] \circ [\![t_{\mathrm{true}}]\!] \circ \mathcal{V}[\![e_2 : \tau]\!] & \\ \quad = & [\![t_1]\!] \circ [\![t_{\mathrm{true}}]\!] \circ [\![t_2]\!] \circ \mathcal{V}[\![v : \tau]\!] & \text{as noted} \\ \quad = & [\![t_2 + t_{\mathrm{true}} + t_1]\!] \circ \mathcal{V}[\![v : \tau]\!] & \end{array}$$

Case $\dfrac{e_1 \stackrel{t_1}{\Rightarrow}_{\mathrm{v}} \texttt{false} \quad e_3 \stackrel{t_3}{\Rightarrow}_{\mathrm{v}} v}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \stackrel{t_3+t_{\mathrm{false}}+t_1}{\Rightarrow}_{\mathrm{v}} v}$**:** Similar to the $e_1 \stackrel{t_1}{\Rightarrow}_{\mathrm{v}} \texttt{true}$ case.

Case $\dfrac{[\texttt{rec}\ x.e/x]e \overset{t}{\Rightarrow}_{\mathrm{v}} v}{\texttt{rec}\ x.e \overset{t+t_{\mathrm{rec}}}{\Rightarrow}_{\mathrm{v}} v}$**:**

As $\vdash \texttt{rec}\ x.e : \tau$, we know that $x : \tau \vdash e : \tau$. Therefore

$$\begin{array}{rcll}
\mathcal{V}[\![\texttt{rec}\ x.e : \tau]\!] & = & \mathsf{fixp}([\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![x : \tau \vdash e : \tau]\!]) & \\
& = & [\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![x : \tau \vdash e : \tau]\!] & \text{Fixed point property} \\
& & \quad \circ \langle \mathsf{id}, \mathsf{fixp}([\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![x : \tau \vdash e : \tau]\!]) \rangle & \\
& = & [\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![x : \tau \vdash e : \tau]\!] \circ \langle \mathsf{id}, \mathcal{V}[\![\texttt{rec}\ x.e : \tau]\!] \rangle & \\
& = & [\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![[\texttt{rec}\ x.e/x]e : \tau]\!] & \text{Substitution Lemma} \\
& = & [\![t_{\mathrm{rec}}]\!] \circ [\![t]\!] \circ \mathcal{V}[\![v : \tau]\!] & \text{induction hypothesis} \\
& = & [\![t + t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![v : \tau]\!] &
\end{array}$$

□

3.4.5 Soundness of the extensional semantics

Corollary 3.4.10 *Suppose that $\vdash e : \tau$, $e \Rightarrow_{\mathrm{v}} v$, and $\mathcal{C}^{\mathrm{V}}[\![c]\!]$ is sound for each constant c. Then*

$$\mathcal{V}_{\mathrm{E}}[\![e : \tau]\!] = \mathcal{V}_{\mathrm{E}}[\![v : \tau]\!]$$

Proof. Assume that $\vdash e : \tau$ and that $e \Rightarrow_{\mathrm{v}} v$. By the construction of the intensional operational semantics, it is clear that for some cost t, $e \overset{t}{\Rightarrow}_{\mathrm{v}} v$. Therefore by the soundness of the intensional semantics, $\mathcal{V}[\![e : \tau]\!] = [\![t]\!] \circ \mathcal{V}[\![v : \tau]\!]$. Thus

$$\mathcal{V}_{\mathrm{E}}[\![e : \tau]\!] = E\mathcal{V}[\![e : \tau]\!] = E([\![t]\!] \circ \mathcal{V}[\![v : \tau]\!]) = \mathcal{V}_{\mathrm{E}}[\![v : \tau]\!]$$

□

3.5 Adequacy of the intensional semantics

Given the separability and adequacy properties of the cost structure, we can prove that the intensional semantics is adequate using the knowledge that the extensional semantics is adequate.

Theorem 3.5.1 *For any closed expression e of type τ, $\mathcal{V}[\![e : \tau]\!] \neq \bot$ if and only if there exists a closed value v of type τ and a $t \in T$ such that $e \overset{t}{\Rightarrow}_{\mathrm{v}} v$.*

Proof. Suppose that $\mathcal{V}[\![e : \tau]\!] \neq \bot$. Then by the adequacy properties of the cost structure,

$$E\mathcal{V}[\![e : \tau]\!] = \mathcal{V}_{\mathrm{E}}[\![e : \tau]\!] \neq \bot$$

Therefore, by the adequacy of the extensional semantics there exists a closed value v of type τ such that $e \Rightarrow_{\mathrm{v}} v$. Therefore there must also exist a $t \in T$ such that $e \overset{t}{\Rightarrow}_{\mathrm{v}} v$.

Now suppose there exists a closed value v of type τ and a $t \in T$ such that $e \overset{t}{\Rightarrow}_{\mathrm{v}} v$. By Lemma 3.4.8 there exists a morphism $y : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ such that $\mathcal{V}[\![v : \tau]\!] = \eta \circ y$. Therefore by soundness $\mathcal{V}[\![e : \tau]\!] = [\![t]\!] \circ \eta \circ g$ so $E(\mathcal{V}[\![e : \tau]\!]) = \mathsf{up} \circ Ey$. By the properties of strictness we thus know that $E(\mathcal{V}[\![e : \tau]\!]) \neq \bot$, therefore, by the adequacy assumptions $\mathcal{V}[\![e : \tau]\!] \neq \bot$ as well. □

3.6 Creating cost structures

In this section we show that there are non-trivial examples of strong cost structures. We do this in five stages. First, we introduce a helpful auxiliary notion, that of *arrow cost structures*. We then show that given a non-trivial arrow cost structure we can create a cost structure. We also extend the definition of proper and strong to cover arrow cost structures and show that if the arrow cost structure is proper or strong, then its associated cost strong is also proper or strong, respectively. Next, we show that for certain kinds of objects, we can use products to form strong arrow cost structures. Lastly, we will show that **PDom** has the necessary properties, including the special objects needed, to build non-trivial cost structures.

The first structure we define, the arrow cost structure, describes what it means to add cost to an object but does not assume that the object with cost is strict. It also does not include the functor E as a method for removing cost, although it does include a collection of morphisms which remove external cost. We use the term arrow cost structure because the cost structure constructed from the arrow cost structure uses the arrow category over $\mathcal{C}$. The arrow category has sufficient structure for us to define the cost-removing function E based on the cost information in the arrow cost structure. For strictness, the arrow cost structure includes some ability to interact with the lifting monad, so we can use the lifting monad (in the arrow category) to add strictness and ensure the necessary adequacy properties.

3.6.1 Arrow categories

The general difficulty found when trying to recover the external semantics from the internal semantics is the need to remove internal costs. The internal costs relating to higher-order types are particularly difficult to remove because the first argument of the exponentiation functor is contravariant. Using the arrow category, we can maintain the extensional semantics so that it can be easily recovered from the intensional semantics.

Given a category $\mathcal{C}$, its *arrow category*, $\mathcal{C}^{\rightarrow}$, has as objects morphisms in $\mathcal{C}$. Given two morphisms $p : X \rightarrow D$ and $p' : X' \rightarrow D'$ in $\mathcal{C}$, a morphism from p to p' in $\mathcal{C}^{\rightarrow}$ is a pair of morphisms (h, d), where $h : X \rightarrow X'$ and $d : D \rightarrow D'$ and

$$p' \circ h = d \circ p_1$$

For our purposes, an object in the arrow category, $p : X \rightarrow D$, represents the projection from the intensional meaning of a type to its extensional meaning. A morphism in the arrow category is thus a pair (h, d) where h is the intensional meaning and d is the extensional meaning. With expressions as morphisms the requirement $p' \circ h = d \circ p$ is equivalent to saying that, if we take the intensional meaning of an expression and then convert the result to the extensional version, we get the same meaning as we would if we converted the input (environment) first and then evaluated the expression extensionally; i.e., the extensional behavior of the intensional semantics is the same as the extensional semantics. Extracting the extensional semantics is then merely a matter of using only the extensional parts of the objects and morphisms. Let E be the functor from $\mathcal{C}^{\rightarrow}$ to $\mathcal{C}$ where for any object $p : X \rightarrow D$ in $\mathcal{C}^{\rightarrow}$, $Ep = D$ and for any morphism $(h, d) : p \rightarrow p'$, $E(h, d) = d$. It is clear then that if d represents the extensional semantics, then E converts an intensional semantics into an extensional one.

That E is a representation follows from the structure of the arrow category itself. Any object A in $\mathcal{C}$ can be turned into an object id_A in $\mathcal{C}^{\rightarrow}$; in our semantics these correspond to types that have trivial internal costs, such as the boolean type. Also, given any n-ary functor $F : \mathcal{C} \times \ldots \times \mathcal{C} \rightarrow \mathcal{C}$, we can derive an n-ary functor $F^{\rightarrow} : \mathcal{C}^{\rightarrow} \times \ldots \times \mathcal{C}^{\rightarrow} \rightarrow \mathcal{C}^{\rightarrow}$, where $F^{\rightarrow}(p_1, \ldots, p_n) = F(p_1, \ldots, p_n)$ and $F^{\rightarrow}((h_1, d_1), \ldots, (h_n, d_n)) = (F(h_1, \ldots, h_n), F(d_1, \ldots, d_n))$. Furthermore, natural transformations convert to natural transformations and universal constructions such as $\langle -, - \rangle$ convert to universal constructions. For example, if $p_1 : X_1 \rightarrow D_1$ and $p_2 : D_1 \rightarrow D_2$ are objects in $\mathcal{C}^{\rightarrow}$, then their product is $p_1 \times p_2 : X_1 \times X_2 \rightarrow D_1 \times D_2$ in $\mathcal{C}^{\rightarrow}$, with projections (π_1, π_1) and (π_2, π_2) and with $\langle (h_1, d_2), (h_2, d_2) \rangle = (\langle h_1, h_2 \rangle, \langle d_1, d_2 \rangle)$. The natural transformation β is converted to a natural transformation $\beta^{\rightarrow}$ in $\mathcal{C}^{\rightarrow}$, where $\beta^{\rightarrow}_{p_1,p_2} = (\beta_{X_1,X_2}, \beta_{D_1,D_2})$; additionally, $\beta^{\rightarrow}_{p_2,p_1} \circ \beta^{\rightarrow}_{p_1,p_2} = \mathsf{id}_{p_1 \times p_2}$.

Lastly, if $\mathcal{C}$ is a **PDom**-enriched category, so is $\mathcal{C}^{\rightarrow}$, where morphisms are ordered pairwise. An object $p : X \rightarrow D$ in $\mathcal{C}^{\rightarrow}$ is strict if and only if X and D are both strict and $\perp_p = (\perp_X, \perp_D)$.

Unlike the covariant functions just listed, functors with contravariant arguments (like those relating to exponentiation) are not so easily converted; to define a projection from $[X_1 \Rightarrow X_2]$ to $[D_1 \Rightarrow D_2]$ we need a morphism from D_1 to X_1. In this case, however, there is no particular reason why the exponentiation object has to be a morphism from $[X_1 \Rightarrow X_2]$ to $[D_1 \Rightarrow D_2]$. Instead we look at the hom-sets, which the exponentiation objects are supposed to reflect. For $p_1 : X_1 \rightarrow D_1$ and $p_2 : X_2 \rightarrow D_2$, $\mathrm{Hom}(p_1, p_2)$ is a *subset* of $\mathrm{Hom}(X_1, X_2) \times \mathrm{Hom}(D_1, D_2)$. Therefore we would expect the exponentiation of p_1 and p_2 to be related to a "subset" of $[X_1 \Rightarrow X_2] \times [D_1 \Rightarrow D_2]$. One method of modeling subsets in category theory uses a morphism (frequently a monomorphism); a "subset" of $[X_1 \Rightarrow X_2] \times [D_1 \Rightarrow D_2]$ would then be a morphism $\iota : B \rightarrow [X_1 \Rightarrow X_2] \times [D_1 \Rightarrow D_2]$. Furthermore, as ι's codomain is a product, we can also describe it as a pair of morphisms, $\iota^X : B \rightarrow [X_1 \Rightarrow X_2]$ and $\iota^D : B \rightarrow [D_1 \Rightarrow D_2]$. The exponentiation object then becomes an example of a common categorical concept: a *pullback*.

Definition 3.6.1 Given a pair of morphisms $f : B \rightarrow A$ and $g : D \rightarrow A$, their pullback is an object P plus a pair of morphisms $p : P \rightarrow B$ and $q : P \rightarrow D$ such that $f \circ p = g \circ q$ and given any other object C with morphisms $h : C \rightarrow B$ and $k : C \rightarrow D$ such that $f \circ h = g \circ k$, there is a unique morphism $r : C \rightarrow P$ such that the following diagram commutes:

$$\begin{tikzcd}
C \arrow[dr, "r"] \arrow[drr, "k"] \arrow[ddr, "h"'] & & \\
& P \arrow[r, "q"'] \arrow[d, "p"] & D \arrow[d, "g"] \\
& B \arrow[r, "f"'] & A
\end{tikzcd}$$

Theorem 3.6.1 *Given a Cartesian closed category $\mathcal{C}$, suppose that for any pair of morphisms*

$$p_1 : X_1 \rightarrow D_1 \quad \textit{and} \quad p_2 : X_2 \rightarrow D_2$$

the pair of morphisms

$$[\mathsf{id} \Rightarrow p_2] : [X_1 \Rightarrow X_2] \rightarrow [X_1 \Rightarrow D_2] \quad \textit{and} \quad [p_1 \Rightarrow \mathsf{id}] : [D_1 \Rightarrow D_2] \rightarrow [X_1 \Rightarrow D_2]$$

always has a pullback. Then $\mathcal{C}^{\rightarrow}$ is Cartesian closed.

Proof. For the morphisms $p_1 : X_1 \rightarrow X_2$ and $p_2 : D_1 \rightarrow D_2$, let the object P_{p_1,p_2} and the morphisms, $\iota^X_{p_1,p_2} : P_{p_1,p_2} \rightarrow [X_1 \Rightarrow X_2]$ and $\iota^D_{p_1,p_2} : P_{p_1,p_2} \rightarrow [D_1 \Rightarrow D_2]$, be the pullback. Then

$\iota^D_{p_1,p_2}$ is the exponentiation of p_1 and p_2 in $\mathcal{C}^{\rightarrow}$. Furthermore, $\mathsf{app}_{p_1,p_2} = (\mathsf{uncurry}(\iota^X_{p_1,p_2}), \mathsf{app}_{D_1,D_2})$ and if $(h,d) : p \times p_1 \rightarrow p_2$, where $p : X \rightarrow D$, then $\mathsf{curry}(h,d) = (r, \mathsf{curry}(d))$, where r is the unique morphism generated by the pullback in the following diagram:

$$\begin{tikzcd}
X \arrow[dr, "r"] \arrow[drr, "\mathsf{curry}(d) \circ p"] \arrow[ddr, "\mathsf{curry}(h)"'] & & \\
& P_{p_1,p_2} \arrow[r, "\iota^D_{p_1,p_2}"'] \arrow[d, "\iota^X_{p_1,p_2}"] & {[D_1 \Rightarrow D_2]} \arrow[d, "{[p_1 \Rightarrow \mathsf{id}]}"] \\
& {[X_1 \Rightarrow X_2]} \arrow[r, "{[\mathsf{id} \Rightarrow p_2]}"'] & {[X_1 \Rightarrow D_2]}
\end{tikzcd}$$

It is straightforward to show that these definitions satisfy all the requirements of a Cartesian closed category. □

The category **PDom** has such pullbacks; in fact, it has all pullbacks. Therefore $\mathbf{PDom}^{\rightarrow}$ is Cartesian closed.

Because $\mathcal{C}^{\rightarrow}$ has many of the same properties as $\mathcal{C}$ (such as Cartesian closure), we can ignore the fact that we are operating in the arrow category when we look at specific examples. A morphism can be treated as a simple morphism; the fact that it is "actually" a pair of morphisms is of interest only when we want to extract the extensional semantics.

3.6.2 Cost structures in the arrow category

If we set $\mathcal{C}^{\mathrm{I}}$ to $\mathcal{C}^{\rightarrow}$, then, by the meaning we gave to the objects of $\mathcal{C}^{\rightarrow}$, we would expect that given $p : X \rightarrow D$, then Cp would have the form $KX \rightarrow LD$ (because EC is L). If we are starting with an arrow cost structure in $\mathcal{C}$, then we would expect that K would be derived from the elements of the cost structure.

Let $(A, \eta^A, (-)^\star, \psi^A, [\![-]\!]^A)$ be defined similarly to the first five elements of a cost structure, except on $\mathcal{C}$. Then A is a function on objects of $\mathcal{C}$, for each object X in $\mathcal{C}$, $\eta^A_X : X \rightarrow AX$ and $[\![t]\!]^A_X : AX \rightarrow AX$, for any morphism $f : X \rightarrow AX'$ in $\mathcal{C}$, $(f)^\star : AX \rightarrow AX'$. Because we assume that A does not handle strictness itself, we need to add strictness properties in order to form K. If we set K to LA, then both KX and $p : KX \rightarrow LD$ will be strict, as desired.

In order for the constructed cost structure to be well-defined, we need to add two elements to the arrow cost structure. First, given that we will be creating morphisms of the form $LAX \rightarrow LD$ (from morphisms of the form $X \rightarrow D$), we need a collection of morphisms δ such that for any object X, $\delta_A : AX \rightarrow X$. These morphisms strip away the external costs. The requirements of δ are needed to ensure that all the morphisms we will be defining are valid; i.e., that removing external costs can be done before or after a cost-structure related operation without affecting the extensional value. These properties are:

- $\delta_X \circ \eta^A_X = \mathsf{id}_X$ – adding 0 cost and then removing it has no effect
- $\delta_{X'} \circ (f)^\star = \delta_{X'} \circ f \circ \delta_{X'}$ – removing the cost before and after the application of $f : X \rightarrow AX'$ is the same as simply removing the cost afterwards

- $\delta_{X_1 \times X_2} \circ \psi^A_{X_1,X_2} = \delta_{X_1} \times \delta_{X_2}$ – removing combined costs is the same as removing the uncombined costs
- $\delta_X \circ [\![t]\!]^A_X = \delta_X$ – adding a constant cost had no effect when costs are removed directly afterwards

We also need a method for controlling the interaction between L and A. This problem typically appears when combining monads (or monad-like structures). There are three possible natural transformations that tend to be used (see [18]). For this paper we use $\mathsf{rd}_X : ALAX \to LAX$. This morphism has the effect of combining costs (similar to $(\mathsf{id})^\star$), but takes into account that one cost may be $\bot$.

To obtain a generic cost structure, we do not require that rd be a natural transformation. All we require is that $\mathsf{rd}_X \circ [\![t]\!]^A_{LAX} \circ \eta^A_{LAX} = L[\![t]\!]^A_X$, which formalizes the notion that rd combines costs, and that $L\delta_X \circ \mathsf{rd}_X = L\delta_X \circ \delta_{LAX}$, which states that we can remove costs either before or after they are combined. The first condition, with t set to 0, is the same as one of the properties listed in [18].

To get a proper or strong arrow cost structure we will have to add additional conditions to rd. All of these conditions can be found in [18].

Definition 3.6.2 Let T be a monoid set and $\mathcal{C}$ be a Cartesian closed, **PDom**-enriched category with lifts, conditionals, and pullbacks. Then an *arrow cost structure of T on $\mathcal{C}$* is a septuple $(A, \eta^A, (-)^\star, \psi^A, [\![-]\!]^A, \delta, \mathsf{rd})$, where

- A is a function on objects of $\mathcal{C}$
- For any object X, $\eta^A_X : X \to AX$
- For any morphism $f : X \to AX'$, $(f)^\star : AX \to AX'$
- For any objects X_1, X_2, $\psi^A_{X_1,X_2} : AX_1 \times AX_2 \to A(X_2 \times X_2)$
- For any object X and any cost $t \in T$, $[\![t]\!]^A_X : AX \to AX$
- For any object X, $\delta_X : AX \to X$
- For any object X, $\mathsf{rd}_X : ALAX \to LAX$.

and the following properties hold:

1. For all objects X
$$[\![0]\!]^A_X = \mathsf{id}_{AX}$$
2. For all objects X and all $t_1, t_2 \in T$,
$$[\![t_2]\!]^A_X \circ [\![t_1]\!]^A_X = [\![t_1 + t_2]\!]^A_X$$
3. For all $f : X \to AX'$ and all $t \in T$,
$$(f)^\star \circ [\![t]\!]^A_X \circ \eta^A_X = [\![t]\!]^A_{X'} \circ f$$
4. For all $t_1, t_2 \in T$ and pairs of objects X_1 and X_2,
$$\psi^A_{X_1,X_2} \circ ([\![t_1]\!]^A_{X_1} \times [\![t_2]\!]^A_{X_2}) \circ (\eta^A_{X_1} \times \eta^A_{X_2}) = [\![t_2 + t_1]\!]^A_{X_1 \times X_2} \circ \eta^A_{X_1 \times X_2}$$

5. For all $t_1, t_2 \in T$ and all $y_1, y_2 : \mathbf{1} \to X$, if $[\![t_1]\!]^A_X \circ \eta^A_X \circ y_1 \leq [\![t_2]\!]^A_X \circ \eta^A_X \circ y_2$, then $[\![t_1]\!]^A = [\![t_2]\!]^A$ and $y_1 \leq y_2$.

6. For all objects X of $\mathcal{C}$,
$$\mathsf{rd}_X \circ [\![t]\!]^A_{LAX} \circ \eta^A_{LAX} = L[\![t]\!]^A_X$$

7. For all objects X of $\mathcal{C}$,
$$\delta_X \circ \eta^A_X = \mathsf{id}_X$$

8. For all $f : X \to AX'$,
$$\delta'_X \circ (f)^\star = \delta'_X \circ f \circ \delta_X$$

9. For all $t \in T$ and all objects X of $\mathcal{C}$,
$$\delta_X \circ [\![t]\!]^A_X = \delta_X$$

10. For all objects X_1 and X_2 of $\mathcal{C}$,
$$\delta_{X_1 \times X_2} \circ \psi^A_{X_1,X_2} = \delta_{X_1} \times \delta_{X_2}$$

11. For all objects X of $\mathcal{C}$,
$$L\delta_X \circ \mathsf{rd}_X = L\delta_X \circ \delta_{LAX}$$

The new properties (6-11) are shown as diagrams in Figure 3.8.

Definition 3.6.3 An arrow cost structure $(A, \eta^A, (-)^\star, \psi^A, [\![-]\!]^A, \delta, \mathsf{rd})$ is *proper* if $(A, \eta^A, (-)^\star)$ is a Kleisli triple (i.e., $(A, \eta^A, (\mathsf{id})^\star)$ is a monad) and the following properties hold:

- For all morphisms $f : X \to AX'$,
$$\mathsf{rd}_{X'} \circ (\eta^A_{LAX'} \circ \mathsf{up}_{AX'} \circ f)^\star = \mathsf{up}_{AX'} \circ f^\star$$

- For all morphisms $f : X \to LAX'$,
$$\mathsf{rd}_{X'} \circ \left(\eta^A_{LAX'} \circ \left(\mathsf{rd}_{X'} \circ (\eta^A_{LAX'} \circ f)^\star\right)^\perp\right)^\star = \left(\mathsf{rd}_{X'} \circ (\eta^A_{LAX'} \circ f)^\star\right)^\perp \circ \mathsf{rd}_X$$

As noted, the extra properties are taken from standard properties for combining monads, although written using Kleisli notation. The properties are a bit cleaner when written using mixed notation. For example, the first property can also be written as
$$\mathsf{rd}_{X'} \circ A\mathsf{up}_{AX'} \circ Af = \mathsf{up}_{AX'} \circ f^\star$$

The Kleisli form, however, more clearly shows the intuition behind the property. The rd transformations combines a cost with a lifted cost and handles the case where the lifted cost is $\perp$. The above property says that if we know the lifted cost is not $\perp$ and that the other cost is 0, then rd as the same result as the lifted cost.

The second property, written in mixed notation, becomes
$$\mathsf{rd}_{X'} \circ A(\mathsf{rd}_{X'} \circ Af)^\perp = (\mathsf{rd}_{X'} \circ Af)^\perp \circ \mathsf{rd}_X$$

With this notation the property is more clearly seen to be related to a naturality requirement on rd.

$$
\begin{tikzcd}
ALAX \arrow[r, "\llbracket t \rrbracket^A"] & ALAX \arrow[d, "\mathsf{rd}"] \\
LAX \arrow[u, "\eta^A"] \arrow[r, "L\llbracket t \rrbracket^A"'] & LAX
\end{tikzcd}
\qquad
\begin{tikzcd}
ALAX \arrow[r, "\mathsf{rd}"] \arrow[d, "\delta"'] & LAX \arrow[d, "L\delta"] \\
LAX \arrow[r, "L\delta"'] & LX
\end{tikzcd}
$$

$$
\begin{tikzcd}
 & AX \arrow[dr, "\delta"] & \\
X \arrow[ur, "\eta^A"] \arrow[rr, "\mathsf{id}"'] & & X
\end{tikzcd}
\qquad
\begin{tikzcd}
AX \arrow[rr, "(f)^\star"] \arrow[d, "\delta"'] & & AX' \arrow[d, "\delta"] \\
X \arrow[r, "f"'] & AX' \arrow[r, "\delta"'] & X'
\end{tikzcd}
$$

$$
\begin{tikzcd}
 & AX \arrow[dr, "\delta"] & \\
AX \arrow[ur, "\llbracket t \rrbracket^A"] \arrow[rr, "\delta"'] & & X
\end{tikzcd}
\qquad
\begin{tikzcd}
 & A(X_1 \times X_2) \arrow[dr, "\delta"] & \\
AX_1 \times AX_2 \arrow[ur, "\psi^A"] \arrow[rr, "\delta \times \delta"'] & & X_1 \times X_2
\end{tikzcd}
$$

Figure 3.8: New properties of arrow cost categories

Definition 3.6.4 An arrow cost structure $(A, \eta^A, (-)^\star, \psi^A, [\![-]\!]^A, \delta, \mathsf{rd})$ is *strong* if it is proper, if $(A, \eta^A, (\mathsf{id})^\star, \tau^A)$ is a strong monad, where $\tau^A = \psi^A \circ (\mathsf{id} \times \eta^A)$, and if for all objects X_1, X_2,

$$\mathsf{rd}_{X_1 \times X_2} \circ AL\tau^A_{X_1,X_2} \circ A\tau^L_{AX_1,X_2} \circ \tau^A_{LAX_1,X_2} = L\tau^A_{X_1,X_2} \circ \tau^L_{AX_1,X_2} \circ (\mathsf{rd}_{X_1} \times \mathsf{id})$$

where $\tau^L = \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up})$.

The added properties is essentially a commutative property between rd, τ^A, and τ^L, i.e., rd and be freely applied before or after a sequence of τ^A and τ^L.

We can now define the constructed cost structure:

Theorem 3.6.2 *If $(A, \eta^A, (-)^\star, [\![-]\!]^A, \psi^A, \delta, \mathsf{rd})$ is an arrow cost structure of T on $\mathcal{C}$, then there exists a cost structure $(C, \eta, (-)^*, [\![-]\!], \psi, E)$ of T on $\mathcal{C}^\to$ over $\mathcal{C}$, where for any $p : X \to D$ and $p' : X' \to D'$,*

- $Cp = Lp \circ L\delta_X : LAX \to LD$,
- $\eta_p = (\mathsf{up}_{AX} \circ \eta^A_X, \mathsf{up}_D)$,
- *For any $(h, d) : p \to Cp'$, $(h, d)^* = ((\mathsf{rd}_{X'} \circ (\eta^A_{LAX'} \circ h)^\star)^\perp, d^\perp)$*
- *For any $t \in T$, $[\![t]\!]_p = (L[\![t]\!]^A_X, \mathsf{id}_{LD})$,*
- $\psi_{p,p'} = (L\psi^A_{X,X'} \circ \mathsf{smash}_{AX,AX'}, \mathsf{smash}_{D,D'})$
- *$Ep = D$, and for any $(h, d) : p \to p'$, $E(h, d) = d$.*

Proof. There are three parts to the proof. We must show that $\mathcal{C}^\to$ has the necessary properties for the intensional category, we must show that all of the morphisms defined are valid, and we must show that all of the properties are defined.

In the previous section we showed that $\mathcal{C}^\to$ is Cartesian closed and **PDom** enriched. It is not difficult to show that $\mathcal{C}^\to$ has lifts and conditionals as well, given that arrow categories preserve functors, universal constructions, and orderings.

Next we show that all the morphisms defined are valid. These follow from the properties given for δ plus the monad properties of lifting. We show that $(h, d)^*$ is valid below; the others are straightforward.

For $(h, d)^*$, because (h, d) is valid, we know that

$$Lp' \circ L\delta \circ h = d \circ p$$

Therefore, by the diagram in Figure 3.9, we know that

$$(Lp' \circ L\delta \circ \mathsf{rd} \circ (\eta^A \circ h)^\star)^\perp = (d \circ p \circ \delta)^\perp$$

that is, that

$$Lp' \circ L\delta \circ (\mathsf{rd} \circ (\eta^A \circ h)^\star)^\perp = d^\perp \circ Lp \circ L\delta$$

Therefore $(h, d)^*$ is valid.

Lastly, we need to show that the cost structure created satisfies all the relevant properties.

$$\begin{tikzcd}
AX \arrow[rrr, "(\eta^A \circ h)^*"] \arrow[dd, "\delta"'] & & & ALAX' \arrow[r, "\mathsf{rd}"] \arrow[dd, "\delta"] & LAX' \arrow[dd, "L\delta"] \\
 & & ALAX' \arrow[dr, "\delta"] & & \\
X \arrow[r, "h"'] \arrow[d, "p"'] & LAX' \arrow[ur, "\eta^A"] \arrow[rr, "\mathsf{id}"'] & & LAX' \arrow[r, "L\delta"'] & LX' \arrow[d, "Lp'"] \\
D \arrow[rrrr, "d"'] & & & & LD'
\end{tikzcd}$$

Figure 3.9: Proof that $(h, d)^*$ is valid

Soundness Properties: The soundness properties follow from the equivalent properties of the arrow cost structure. The most complicated case is the property that

$$(h, d)^* \circ [\![t]\!] \circ \eta = (h, d)$$

which we prove as follows:

Given $p : X \to D$, $p' : X' \to D'$, and $(h, d) : p \to p'$,

$$\begin{array}{ll}
(h, d)^* \circ [\![t]\!]_p \circ \eta_p & \\
\quad = & ((\mathsf{rd}_{X'} \circ (\eta^A_{LAX'} \circ h)^*)^\perp, d^\perp) \circ (L[\![t]\!]^A_X, \mathsf{id}_{LD}) \circ (\mathsf{up}_{AX} \circ \eta^A_X, \mathsf{up}_D) \\
\quad = & ((\mathsf{rd}_{X'} \circ (\eta^A_{LAX'} \circ h)^*)^\perp \circ L[\![t]\!]^A_X \circ \mathsf{up}_{AX} \circ \eta^A_X, d^\perp \circ \mathsf{up}_D)
\end{array}$$

By the monad properties of lifting, the right side, $d^\perp \circ \mathsf{up}_D$, equals d. By the diagram in Figure 3.10 the left side equals $L[\![t]\!]^A_{X'} \circ h$. Thus

$$(h, d)^* \circ [\![t]\!]_p \circ \eta_p = (L[\![t]\!]^A_{X'} \circ h, d) = [\![t]\!]_{p'} \circ (h, d)$$

as required.

Separability properties:

1. E is a representation.
   The method by which functors are lifted to the arrow categories ensures that E is a representation of them. Furthermore, by definition E preserves Cartesian closure and **PDom** enrichedness.
2. $E(C, \eta, (-)^*, \psi)$ is the lifting monad.
   This follows directly from the definition of E and $(C, \eta, (-)^*, \psi)$.
3. For all costs t and objects p, $E[\![t]\!]_p = \mathsf{id}_{Ep}$.
   This follows directly from the definition.

$$\begin{tikzcd}
X \arrow[rr, "h"] \arrow[d, "\eta^A"'] & & LAX' \arrow[d, "\eta^A"] \arrow[dr, "\mathsf{id}"] & \\
AX \arrow[r, "\llbracket t \rrbracket^A"] \arrow[d, "\mathsf{up}"'] & AX \arrow[dr, "(\eta^A \circ h)^\star"'] & ALAX' \arrow[d, "\llbracket t \rrbracket^A"] & LAX' \arrow[ddl, "L\llbracket t \rrbracket^A"] \\
LAX \arrow[d, "L\llbracket t \rrbracket^A"'] \arrow[drr, "(\mathsf{rd} \circ (\eta^A \circ h)^\star \circ \llbracket t \rrbracket^A)^\perp"] & & ALAX' \arrow[d, "\mathsf{rd}"] & \\
LAX \arrow[rr, "(\mathsf{rd} \circ (\eta^A \circ h)^\star)^\perp"'] & & LAX' &
\end{tikzcd}$$

Figure 3.10: Part of proof that soundness properties hold

Adequacy properties:

1. For all objects p, Cp is strict.
   This follows directly from the definition.
2. For all $(h,d) : \mathbf{1} \to Cp$, $(h,d) = \perp$ if and only if $E(h,d) = \perp$.
   Clearly if $(h,d) = \perp = (\perp,\perp)$, then $E(h,d) = d = \perp$. Now suppose that $d = \perp$, but $h \neq \perp$. As $h : \mathbf{1} \to LAX$, by the definition of strictness for categories there exists an $x : \mathbf{1} \to AX$ such that $h = \mathsf{up} \circ x$. Then

   $$Lp \circ L\delta \circ h = Lp \circ L\delta \circ \mathsf{up} \circ x = \mathsf{up} \circ p \circ \delta \circ x$$

   (h,d), however, is a valid morphism in $\mathcal{C}^{\to}$, so $Lp \circ L\delta \circ h = d \circ \mathsf{id}_{\mathbf{1}}$., i.e., $\mathsf{up} \circ p \circ \delta \circ x = \perp$. By the definition, of strictness, however, $\mathsf{up} \circ p \circ \delta \circ x \neq \perp$. Therefore h must be $\perp$, so $(h,d) = \perp$.

Ordering property:

1. For all costs t, t' and all morphisms $(h,d), (h',d') : \mathbf{1} \to p$, if

   $$[\![t]\!] \circ \eta \circ (h,d) \leq [\![t']\!] \circ \eta \circ (h',d')$$

   then $[\![t]\!] = [\![t']\!]$ and $(h,d) \leq (h',d')$.
   First,

   $$[\![t]\!] \circ \eta \circ (h,d) = (L[\![t]\!]^A \circ \mathsf{up} \circ \eta^A \circ h, \mathsf{up} \circ d) = (\mathsf{up} \circ [\![t]\!]^A \circ \eta^A \circ h, \mathsf{up} \circ d)$$

   and

   $$[\![t']\!] \circ \eta \circ (h',d') = (L[\![t']\!]^A \circ \mathsf{up} \circ \eta^A \circ h', \mathsf{up} \circ d') = (\mathsf{up} \circ [\![t']\!]^A \circ \eta^A \circ h', \mathsf{up} \circ d')$$

   Thus we know that $L[\![t]\!]^A \circ \mathsf{up} \circ \eta^A \circ h \leq L[\![t']\!]^A \circ \mathsf{up} \circ \eta^A \circ h'$ and $\mathsf{up} \circ d \leq \mathsf{up} \circ d'$. By the ordering requirements of L, we thus know that $[\![t]\!]^A \circ \eta^A \circ h \leq [\![t']\!]^A \circ \eta^A \circ h'$ and $d \leq d'$. By property 5, $[\![t]\!] = [\![t']\!]$ and $h \leq h'$, so $(h,d) \leq (h',d')$ as well.

$\square$

Theorem 3.6.3 *If an arrow cost structure $(A, \eta^A, (-)^\star, \psi^A, [\![-]\!]^A, \delta, \mathsf{rd})$ is proper, then the derived cost structure $(C, \eta, (-)^*, [\![-]\!], \psi, E)$ from the previous theorem is proper.*
Proof. We need to show that $(C, \eta, (-)^*)$ forms a Kleisli triple. The first condition is already shown; what is needed is to show the other two. Thus let $p : X \to D$, $p' : X' \to D'$, $p'' : X'' \to D''$, $(h,d) : p \to Cp'$, and $(h',d') : p' \to Cp''$. Because we know that the arrow category is proper, we will be freely using both Kleisli notation and monad notation for A in the proofs, such as noting that $(h,d)^* = ((\mathsf{rd} \circ Ah)^\perp, d^\perp)$

- $(\eta_p)^* = \mathsf{id}_{Cp}$

$$\begin{aligned}
\eta^* &= (\mathsf{up} \circ \eta^A, \mathsf{up})^* \\
&= ((\mathsf{rd} \circ A\mathsf{up} \circ A\eta^A)^\perp, \mathsf{up}^\perp) \\
&= ((\mathsf{up} \circ (\eta^A)^\star)^\perp, \mathsf{id}) \qquad \text{first property of } \mathsf{rd} \\
&= ((\mathsf{up})^\perp, \mathsf{id}) \\
&= (\mathsf{id}, \mathsf{id}) \\
&= \mathsf{id}
\end{aligned}$$

- $(h', d')^* \circ (h, d)^* = ((h', d)^* \circ (h, d))^*$

$$\begin{array}{rll}
\multicolumn{2}{l}{((h', d)^* \circ (h, d))^*} & \\
= & (((\mathsf{rd} \circ Ah')^{\perp}, d'^{\perp}) \circ (h, d))^* & \\
= & (((\mathsf{rd} \circ Ah')^{\perp} \circ h, d'^{\perp} \circ d))^* & \\
= & ((\mathsf{rd} \circ A(\mathsf{rd} \circ Ah')^{\perp} \circ Ah)^{\perp}, (d'^{\perp} \circ d)^{\perp}) & \\
= & (((\mathsf{rd} \circ Ah')^{\perp} \circ \mathsf{rd} \circ Ah)^{\perp}, (d'^{\perp} \circ d)^{\perp}) & \text{second property of } \mathsf{rd} \\
= & ((\mathsf{rd} \circ Ah')^{\perp} \circ (\mathsf{rd} \circ Ah)^{\perp}, d'^{\perp} \circ d^{\perp}) & \text{monad property} \\
= & ((\mathsf{rd} \circ Ah')^{\perp}, d'^{\perp}) \circ ((\mathsf{rd} \circ Ah)^{\perp}, d^{\perp}) & \\
= & (h', d')^* \circ (h, d)^* &
\end{array}$$

Thus $(C, \eta, (-)^*)$ is a Kleisli triple, so the cost structure is proper. □

Theorem 3.6.4 *If an arrow cost structure $(A, \eta^A, (-)^\star, \psi^A, [\![-]\!]^A, \delta, \mathsf{rd})$ is strong, then the derived cost structure $(C, \eta, (-)^*, [\![-]\!], \psi, E)$ from the previous theorem is also strong.*

Proof. We already know from Theorem 3.6.3 that $(C, \eta, (-)^*, [\![-]\!], \psi, E)$ is proper, so all we need to do is show that the additional strength properties are met. To do this is will be useful to be able to write τ in terms of τ^A and τ^L rather than ψ. Since

$$\begin{array}{rll}
\tau = & \psi \circ (\mathsf{id} \times \eta) & \\
= & (L\psi^A \circ \mathsf{smash}, \mathsf{smash}) \circ ((\mathsf{id} \times \mathsf{up}) \circ (\mathsf{id} \times \eta^A), \mathsf{id} \times \mathsf{up}) & \\
= & (L\psi^A \circ \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up}) \circ (\mathsf{id} \times \eta^A), \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up})) & \\
= & (L\psi^A \circ \mathsf{smash} \circ (\mathsf{id} \times L\eta^A) \circ (\mathsf{id} \times \mathsf{up}), \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up})) & \\
= & (L\psi^A \circ L(\mathsf{id} \times \eta^A) \circ \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up}), \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up})) & \text{naturality of } \mathsf{smash} \\
= & (L\tau^A \circ \tau^L, \tau^L) &
\end{array}$$

we know that for all morphisms $p_1 : X_1 \to D_1$ and $p_2 : X_2 \to D_2$,

$$\tau_{p_1,p_2} = (L\tau^A_{X_1,X_2} \circ \tau^L_{AX_1,X_2}, \tau^L_{D_1,D_2})$$

Furthermore, because the cost structures are proper, we know that C is a functor and that for any morphism $(h, d) : p \to p'$

$$\begin{array}{rll}
C(h, d) = & (\eta \circ (h, d))^* & \\
= & (\mathsf{up} \circ \eta^A \circ h, \mathsf{up} \circ d)^* & \\
= & ((\mathsf{rd} \circ A\mathsf{up} \circ A\eta^A \circ Ah)^{\perp}, (\mathsf{up} \circ d)^{\perp}) & \\
= & ((\mathsf{up} \circ (\eta^A \circ h)^\star)^{\perp}, Ld) & \mathsf{rd} \text{ property for proper arrow cost structures} \\
= & ((\mathsf{up} \circ Ah)^{\perp}, Ld) & \text{definition of } A \\
= & (LAh, Ld) &
\end{array}$$

We can use these results to prove the three strength properties not already shown:

- $C\pi_1 \circ \tau_{p_1,p_2} = \pi_1$

$$\begin{array}{rll}
C\pi_1 \circ \tau = & (LA\pi_1, L\pi_1) \circ (L\tau^A \circ \tau^L, \tau^L) & \\
= & (LA\pi_1 \circ L\tau^A \circ \tau^L, L\pi_1 \circ \tau^L) & \\
= & (L\pi_1 \circ \tau^L, L\pi_1 \circ \tau^L) & \text{strength property of } \tau^A \\
= & (\pi_1, \pi_1) & \text{strength property of } \tau^L \\
= & \pi_1 &
\end{array}$$

- $C\alpha^r_{p_1,p_2,p_3} \circ \tau_{p_1\times p_2,p_3} \circ (\tau_{p_1,p_2} \times \mathsf{id}_{p_3}) = \tau_{p_1,p_2\times p_3} \circ \alpha^r_{Cp_1,p_2,p_3}$

$$\begin{array}{rll}
\multicolumn{3}{l}{C\alpha^r \circ \tau \circ (\tau \times \mathsf{id})} \\
= & (LA\alpha^r, L\alpha^r) \circ (L\tau^A \circ \tau^L, \tau^L) \circ ((L\tau^A \times \mathsf{id}) \circ (\tau^L \times \mathsf{id}), \tau^L \times \mathsf{id}) & \\
= & (LA\alpha^r \circ L\tau^A \circ \tau^L \circ (L\tau^A \times \mathsf{id}) \circ (\tau^L \times \mathsf{id}), L\alpha^r \circ \tau^L \circ (\tau^L \times \mathsf{id})) & \\
= & (LA\alpha^r \circ L\tau^A \circ L(\tau^A \times \mathsf{id}) \circ \tau^L \circ (\tau^L \times \mathsf{id}), \tau^L \circ \alpha^r) & \text{naturality of } \tau^L \\
= & (L\tau^A \circ L\alpha^r \circ \tau^L \circ (\tau^L \times \mathsf{id}), \tau^L \circ \alpha^r) & \text{strength of } \tau^A \\
= & (L\tau^A \circ \tau^L \circ \alpha^r, \tau^L \circ \alpha^r) & \text{strength of } \tau^L \\
= & \tau \circ \alpha^r &
\end{array}$$

- $(\tau_{p_1',p_2'} \circ ((h_1, d_1) \times (h_2, d_2)))^* \circ \tau_{p_1,p_2} = \tau_{p_1',p_2'} \circ ((h,d)^* \times (h_2, d_2))$

$$\begin{array}{rll}
\multicolumn{3}{l}{(\tau \circ ((h_1, d_1) \times (h_2, d_2)))^* \circ \tau} \\
= & ((L\tau^A \circ \tau^L, \tau^L) \circ (h_1 \times h_2, d_1 \times d_2))^* \circ \tau & \\
= & (L\tau^A \circ \tau^L \circ (h_1 \times h_2), \tau^L \circ (d_1 \times d_2))^* \circ \tau & \\
= & ((\mathsf{rd} \circ A(L\tau^A \circ \tau^L \circ (h_1 \times h_2)))^\perp, (\tau^L \circ (d_1 \times d_2))^\perp) \circ (L\tau^A \circ \tau^L, \tau^L) & \\
= & ((\mathsf{rd} \circ AL\tau^A \circ A\tau^L \circ A(h_1 \times h_2))^\perp \circ L\tau^A \circ \tau^L, (\tau^L \circ (d_1 \times d_2))^\perp \circ \tau^L) & \\
= & ((\mathsf{rd} \circ AL\tau^A \circ A\tau^L \circ A(h_1 \times h_2) \circ \tau^A)^\perp \circ \tau^L, (\tau^L \circ (d_1 \times d_2))^\perp \circ \tau^L) & \\
= & & \text{monad property of } (-)^\perp \\
= & ((\mathsf{rd} \circ AL\tau^A \circ A\tau^L \circ \tau^A \circ (Ah_1 \times h_2))^\perp \circ \tau^L, (\tau^L \circ (d_1 \times d_2))^\perp \circ \tau^L) & \\
= & & \text{naturality of } \tau^A \\
= & ((L\tau^A \circ \tau^L \circ (\mathsf{rd} \times \mathsf{id}) \circ (Ah_1 \times h_2))^\perp \circ \tau^L, (\tau^L \circ (d_1 \times d_2))^\perp \circ \tau^L) & \\
= & & \text{strength of } \mathsf{rd} \\
= & (L\tau^A \circ (\tau^L \circ (\mathsf{rd} \times \mathsf{id}) \circ (Ah_1 \times h_2))^\perp \circ \tau^L, (\tau^L \circ (d_1 \times d_2))^\perp \circ \tau^L) & \\
= & & \text{monad property of } (-)^\perp \\
= & (L\tau^A \circ \tau^L \circ ((\mathsf{rd} \circ Ah_1)^\perp \times h_2), \tau^L \circ ({d_1}^\perp \times d_2)) & \text{strength of } \tau^L \\
= & (L\tau^A \circ \tau^L, \tau^L) \circ ((\mathsf{rd} \circ Ah_1)^\perp \times h_2, {d_1}^\perp \times d_2) & \\
= & \tau \circ (((\mathsf{rd} \circ Ah_1)^\perp, {d_1}^\perp) \times (h_2, d_2)) & \\
= & \tau \circ ((h_1, d_1)^* \times (h_2, d_2)) &
\end{array}$$

$\square$

3.6.3 An example of an arrow cost structure

Suppose that we have an object $\mathbf{T}$ that is a *monoid*; in **Set** a monoid is a set $\mathbf{T}$ with an identity 0 and binary operator $+$ such that $a + 0 = 0 + a = a$ and $a + (b + c) = (a + b) + c$. For categories with products a monoid is an object $\mathbf{T}$ with an identity morphism $\mathsf{e} : \mathbf{1} \to \mathbf{T}$ and a morphism $\mathsf{m} : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$ such that the following diagrams commute:

$$\begin{tikzcd}
\mathbf{T} \arrow[r, "\langle \mathsf{id}, \mathsf{e} \circ ! \rangle"] \arrow[dr, "\mathsf{id}"'] & \mathbf{T} \times \mathbf{T} \arrow[d, "\mathsf{m}"] & \mathbf{T} \arrow[l, "\langle \mathsf{e} \circ !, \mathsf{id} \rangle"'] \arrow[dl, "\mathsf{id}"] \\
 & \mathbf{T} &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbf{T} \times (\mathbf{T} \times \mathbf{T}) \arrow[r, "\alpha^l"] \arrow[d, "\mathsf{id} \times \mathsf{m}"'] & (\mathbf{T} \times \mathbf{T}) \times \mathbf{T} \arrow[r, "\mathsf{m} \times \mathsf{id}"] & \mathbf{T} \times \mathbf{T} \arrow[d, "\mathsf{m}"] \\
\mathbf{T} \times \mathbf{T} \arrow[rr, "\mathsf{m}"'] & & \mathbf{T}
\end{tikzcd}$$

Furthermore, assume that for each cost t, there is a morphism $[\![t]\!] : \mathbf{1} \to \mathbf{T}$ such that $[\![0]\!] = \mathsf{e}$, $\mathsf{m} \circ \langle [\![t_1]\!], [\![t_2]\!] \rangle = [\![t_1 + t_2]\!]$ and the set $\{[\![t]\!] \mid t \in T\}$ is discretely ordered. These requirements are trivially satisfied for any monoid by setting $[\![t]\!]$ to e. In **PDom** the set T itself, with the discrete ordering, is a non-trivial example, where $[\![t]\!] = t$ and $\mathsf{m}(t_1, t_2) = t_1 + t_2$.

From the above monoid we can define an arrow cost structure $(\mathbf{T}, \eta^{\mathrm{T}}, (-)^{\star}, \psi^{\mathrm{T}}, [\![-]\!]^{\mathrm{T}}, \delta^{\mathrm{T}}, \mathsf{reduce})$. First, let $\mathbf{T}X$ be $X \times \mathbf{T}$, i.e., an object with cost is simply an object paired with an element of $\mathbf{T}$. Therefore $\eta^{\mathrm{T}}_X : X \to X \times \mathbf{T}$ pairs X with the 0 cost element represented by e. For a given morphism $f : X \to X'$, its lift, $(f)^{\star} : X \times \mathbf{T} \to X' \times \mathbf{T}$, applies f to part of the product without cost and then uses m to combine the cost object from f and the original cost object. For objects X_1, X_2, the morphism $\psi^{\mathrm{T}}_{X_1,X_2} : (X_1 \times \mathbf{T}) \times (X_2 \times \mathbf{T}) \to (X_1 \times X_2) \times \mathbf{T}$ uses m to combine the two cost objects. The morphism $[\![t]\!]^{\mathrm{T}}_X : X \times \mathbf{T} \to X \times \mathbf{T}$ uses $[\![t]\!]$ to create a cost object that represents the constant cost t and then uses m to combine it with the original cost object. The morphism $\delta^{\mathrm{T}}_X : X \times \mathbf{T} \to X$ simply uses project to remove the cost object. Lastly the morphism $\mathsf{reduce}_X : L(X \times \mathbf{T}) \times \mathbf{T} \to L(X \times \mathbf{T})$ uses smash to handle $\perp$ and $L\mathsf{m}$ to combine the two cost objects in non-$\perp$ cases. Most of the definitions also need to include various product isomorphisms needed to regroup the objects so that e, m and $[\![t]\!]$ are applicable. The actual definitions are thus as follows:

$$
\begin{aligned}
\mathbf{T}X &= X \times \mathbf{T} \\
\eta^{\mathrm{T}}_X &= \langle \mathsf{id}_X, \mathsf{e} \circ !_X \rangle \\
f^{\star} &= (\mathsf{id}_{X'} \times \mathsf{m}) \circ \alpha^r_{X',\mathbf{T},\mathbf{T}} \circ (f \times \mathsf{id}_{\mathbf{T}}) \\
\psi^{\mathrm{T}}_{X_1,X_2} &= (\mathsf{id}_{X_1 \times X_2} \times \mathsf{m}) \circ (\mathsf{id}_{X_1 \times X_2} \times \beta_{\mathbf{T},\mathbf{T}}) \circ \phi_{X_1,\mathbf{T},X_2,\mathbf{T}} \\
[\![t]\!]^{\mathrm{T}}_X &= (\mathsf{id}_X \times \mathsf{m}) \circ (\mathsf{id}_X \times \langle \mathsf{id}_{\mathbf{T}}, [\![t]\!] \circ !_{\mathbf{T}} \rangle) \\
\delta^{\mathrm{T}}_X &= \pi_1 \\
\mathsf{reduce}_X &= L(\mathsf{id}_X \times \mathsf{m}) \circ L\alpha^r_{X,\mathbf{T},\mathbf{T}} \circ \mathsf{smash}_{X \times \mathbf{T},\mathbf{T}} \circ (\mathsf{id}_{L(X \times \mathbf{T})} \times \mathsf{up}_{\mathbf{T}}) \\
&= L(\mathsf{id}_X)^{\star} \circ \mathsf{smash}_{X \times \mathbf{T},\mathbf{T}} \circ (\mathsf{id}_{L(X \times \mathbf{T})} \times \mathsf{up}_{\mathbf{T}})
\end{aligned}
$$

This is a valid strong arrow cost structure, as seen below:

Theorem 3.6.5 *If $\mathcal{C}$ is a Cartesian closed,* **PDom***-enriched category with pullbacks, and* $\mathbf{T}$ *is a monoid object with* $[\![-]\!]$*, then the sextuple* $(\mathbf{T}, \eta^{\mathrm{T}}, (-)^{\star}, \psi^{\mathrm{T}}, [\![-]\!]^{\mathrm{T}}, \delta^{\mathrm{T}}, \mathsf{reduce})$ *is a strong arrow cost structure.*

Proof. The eleven properties for an arrow cost structure hold as follows:

Prop. #1: For all objects X, $[\![0]\!]^{\mathrm{T}}_X = \mathsf{id}_{TX}$:

$$
\begin{aligned}
[\![0]\!]^{\mathrm{T}} &= (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{id}, [\![0]\!] \circ ! \rangle) \\
&= (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{id}, \mathsf{e} \circ ! \rangle) \\
&= \mathsf{id} && \text{monoid property}
\end{aligned}
$$

Prop. #2: For all objects X and all $t_1, t_2 \in T$, $[\![t_2]\!]^{\mathrm{T}}_X \circ [\![t_1]\!]^{\mathrm{T}}_X = [\![t_1 + t_2]\!]^{\mathrm{T}}_X$:

$[\![t_2]\!]^{\mathrm{T}}_X \circ [\![t_1]\!]^{\mathrm{T}}_X$ expands out to

$$(\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{id}, [\![t_2]\!] \circ ! \rangle) \circ (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{id}, [\![t_1]\!] \circ ! \rangle)$$

$$
\begin{tikzcd}
X \times \mathbf{T} \arrow[r, "\mathsf{id} \times \langle \mathsf{id}, ! \rangle"] \arrow[d, "\mathsf{id} \times \langle \mathsf{id}, ! \rangle"'] & X \times (\mathbf{T} \times \mathbf{1}) \arrow[d, "\mathsf{id} \times (\mathsf{id} \times \langle [\![t_1]\!], [\![t_2]\!] \rangle)"'] \arrow[dr, "\mathsf{id} \times (\mathsf{id} \times [\![t_1 + t_2]\!])"] & \\
X \times (\mathbf{T} \times \mathbf{1}) \arrow[r, "\mathsf{id} \times (\mathsf{id} \times \langle [\![t_1]\!], [\![t_2]\!] \rangle)"'] \arrow[d, "\mathsf{id} \times (\mathsf{id} \times [\![t_1]\!])"'] & X \times (\mathbf{T} \times (\mathbf{T} \times \mathbf{T})) \arrow[r, "\mathsf{id} \times (\mathsf{id} \times \mathsf{m})"'] \arrow[d, "\mathsf{id} \times \alpha^l"'] & X \times (\mathbf{T} \times \mathbf{T}) \arrow[dd, "\mathsf{id} \times \mathsf{m}"] \\
X \times (\mathbf{T} \times \mathbf{T}) \arrow[r, "\mathsf{id} \times \langle \mathsf{id}, [\![t_2]\!] \circ ! \rangle"'] \arrow[d, "\mathsf{id} \times \mathsf{m}"'] & X \times ((\mathbf{T} \times \mathbf{T}) \times \mathbf{T}) \arrow[d, "\mathsf{id} \times (\mathsf{m} \times \mathsf{id})"'] & \\
X \times \mathbf{T} \arrow[r, "\mathsf{id} \times \langle \mathsf{id}, [\![t_2]\!] \circ ! \rangle"'] & X \times (\mathbf{T} \times \mathbf{T}) \arrow[r, "\mathsf{id} \times \mathsf{m}"'] & X \times \mathbf{T}
\end{tikzcd}
$$

Figure 3.11: Proof of property #2

and $[\![t_1 + t_2]\!]^{\mathrm{T}}_X$ expands out to

$$(\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{id}, [\![t_1 + t_2]\!] \circ ! \rangle)$$

The diagram in Figure 3.11 shows that these two expressions are equal.

Prop. #3: For all $f : X \to \mathbf{T}X'$ and all $t \in T$, $f^\star \circ [\![t]\!]^{\mathrm{T}}_X \circ \eta^{\mathrm{T}}_X = [\![t]\!]^{\mathrm{T}}_{X'} \circ f$:

$f^\star \circ [\![t]\!]^{\mathrm{T}}_X \circ \eta^{\mathrm{T}}_X$ expands to

$$(\mathsf{id} \times \mathsf{m}) \circ \alpha^r \circ (f \times \mathsf{id}) \circ (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{id}, [\![t]\!] \rangle \circ !) \circ \langle \mathsf{id}, \mathsf{e} \circ ! \rangle$$

and $[\![t]\!]^{\mathrm{T}}_{X'} \circ f$ expands to

$$(\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{id}, [\![t]\!] \circ ! \rangle) \circ f$$

The diagram in Figure 3.12 shows that the two are equal.

Prop. #4: For all objects X_1 and X_2 and all $t_1, t_2 \in T$,

$$\psi^{\mathrm{T}}_{X_1, X_2} \circ ([\![t_1]\!]^{\mathrm{T}}_{X_1} \times [\![t_2]\!]^{\mathrm{T}}_{X_2}) \circ (\eta^{\mathrm{T}}_{X_1} \times \eta^{\mathrm{T}}_{X_2}) = [\![t_2 + t_1]\!]^{\mathrm{T}}_{X_1 \times X_2} \circ \eta^{\mathrm{T}}_{X_1 \times X_2}$$

First note that for any $t \in T$,

$$\begin{aligned}
\mathsf{m} \circ \langle \mathsf{id}, [\![t]\!] \circ ! \rangle \circ \mathsf{e} &= \mathsf{m} \circ \langle \mathsf{e}, [\![t]\!] \circ !_1 \rangle \\
&= \mathsf{m} \circ \langle [\![0]\!], [\![t]\!] \rangle \\
&= [\![0 + t]\!] \\
&= [\![t]\!]
\end{aligned}$$

$$
\begin{tikzcd}
X \arrow[r, "\langle \mathsf{id}, \mathsf{e} \circ ! \rangle"] \arrow[d, "f"'] & X \times \mathbf{T} \arrow[r, "\mathsf{id} \times \langle \mathsf{id}, \llbracket t \rrbracket \circ ! \rangle"] \arrow[d, "f \times \mathsf{id}"'] & X \times (\mathbf{T} \times \mathbf{T}) \arrow[r, "\mathsf{id} \times \mathsf{m}"] \arrow[d, "f \times \mathsf{id}"] & X \times \mathbf{T} \arrow[d, "f \times \mathsf{id}"] \\
X' \times \mathbf{T} \arrow[r, "\langle \mathsf{id}, \mathsf{e} \circ ! \rangle"] \arrow[d, "\mathsf{id}"'] & (X' \times \mathbf{T}) \times \mathbf{T} \arrow[r, "\mathsf{id} \times \langle \mathsf{id}, \llbracket t \rrbracket \circ ! \rangle"] \arrow[d, "\alpha^r"'] & (X' \times \mathbf{T}) \times (\mathbf{T} \times \mathbf{T}) \arrow[r, "\mathsf{id} \times \mathsf{m}"] \arrow[d, "\alpha^r"] & (X' \times \mathbf{T}) \times \mathbf{T} \arrow[d, "\alpha^r"] \\
X' \times \mathbf{T} \arrow[r, "\mathsf{id} \times \langle \mathsf{id}, \mathsf{e} \circ ! \rangle"] \arrow[dr, "\mathsf{id}"'] & X' \times (\mathbf{T} \times \mathbf{T}) \arrow[r, "\mathsf{id} \times (\mathsf{id} \times \langle \mathsf{id}, \llbracket t \rrbracket \circ ! \rangle)"] \arrow[d, "\mathsf{id} \times \mathsf{m}"] & X' \times (\mathbf{T} \times (\mathbf{T} \times \mathbf{T})) \arrow[r, "\mathsf{id} \times (\mathsf{id} \times \mathsf{m})"] \arrow[d, "\mathsf{id} \times \alpha^l"] & X' \times (\mathbf{T} \times \mathbf{T}) \arrow[d, "\mathsf{id} \times \mathsf{m}"] \\
 & X' \times \mathbf{T} \arrow[dr, "\mathsf{id} \times \langle \mathsf{id}, \llbracket t \rrbracket \circ ! \rangle"'] & X' \times ((\mathbf{T} \times \mathbf{T}) \times \mathbf{T}) \arrow[d, "\mathsf{id} \times (\mathsf{m} \times \mathsf{id})"] & X' \times \mathbf{T} \\
 & & X' \times (\mathbf{T} \times \mathbf{T}) \arrow[ur, "\mathsf{id} \times \mathsf{m}"'] &
\end{tikzcd}
$$

Figure 3.12: Proof of property #3

$$\begin{tikzcd}
X_1 \times X_2 \arrow[r, "\langle \mathsf{id}, ! \rangle"] \arrow[d, "{\langle \mathsf{id}, [\![t_1]\!] \circ ! \rangle \atop \times \langle \mathsf{id}, [\![t_2]\!] \circ ! \rangle}"'] & (X_1 \times X_2) \times \mathbf{1} \arrow[rr, "\mathsf{id}"] \arrow[d, "\mathsf{id} \times \langle [\![t_1]\!], [\![t_2]\!] \rangle"'] \arrow[dr, "\mathsf{id} \times \langle [\![t_2]\!], [\![t_1]\!] \rangle"] & & (X_1 \times X_2) \times \mathbf{1} \arrow[d, "\langle \mathsf{id}, [\![t_2 + t_1]\!] \rangle"] \\
(X_1 \times \mathbf{T}) \times (X_2 \times \mathbf{T}) \arrow[r, "\phi"'] & (X_1 \times X_2) \times (\mathbf{T} \times \mathbf{T}) \arrow[r, "\mathsf{id} \times \beta"'] & (X_1 \times X_2) \times (\mathbf{T} \times \mathbf{T}) \arrow[r, "\mathsf{id} \times \mathsf{m}"'] & (X_1 \times X_2) \times \mathbf{T}
\end{tikzcd}$$

Figure 3.13: Proof of property #4

which means that

$$(\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{id}, [\![t]\!] \circ ! \rangle) \circ \langle \mathsf{id}, \mathsf{e} \circ ! \rangle = \langle \mathsf{id}, [\![t]\!] \circ ! \rangle$$

Next note that

$$\begin{array}{ll}
\multicolumn{2}{l}{\psi^{\mathrm{T}} \circ ([\![t_1]\!]^{\mathrm{T}} \times [\![t_2]\!]^{\mathrm{T}}) \circ (\eta^{\mathrm{T}} \times \eta^{\mathrm{T}})} \\
= & (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \beta) \circ \phi \circ ((\mathsf{id} \times \mathsf{m}) \times (\mathsf{id} \times \mathsf{m})) \\
& \quad \circ ((\mathsf{id} \times \langle \mathsf{id}, [\![t_1]\!] \circ ! \rangle) \times (\mathsf{id} \times \langle \mathsf{id}, [\![t_2]\!] \circ ! \rangle)) \\
& \quad \circ (\langle \mathsf{id}, \mathsf{e} \circ ! \rangle \times \langle \mathsf{id}, \mathsf{e} \circ ! \rangle) \\
= & (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \beta) \circ \phi \circ (\langle \mathsf{id}, [\![t_1]\!] \circ ! \rangle \times \langle \mathsf{id}, [\![t_2]\!] \circ ! \rangle)
\end{array}$$

and

$$\begin{array}{rcl}
[\![t_2 + t_1]\!]^{\mathrm{T}} \circ \eta^{\mathrm{T}} & = & [\![t_2 + t_1]\!]^{\mathrm{T}} \circ \langle \mathsf{id}, \mathsf{e} \circ ! \rangle \\
& = & (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{id}, [\![t_2 + t_1]\!] \circ ! \rangle) \circ (\mathsf{id} \times \mathsf{e}) \circ \langle \mathsf{id}, ! \rangle \\
& = & \langle \mathsf{id}, [\![t_2 + t_1]\!] \rangle
\end{array}$$

That the two are equal is shown in Figure 3.13.

Prop. #5: For all costs t_1, t_2, and morphisms $g_1, g_2 : \mathbf{1} \to X$, if $[\![t_1]\!]_X^{\mathrm{T}} \circ \eta_X^{\mathrm{T}} \circ g_1 \leq [\![t_2]\!]_X^{\mathrm{T}} \circ \eta_X^{\mathrm{T}} \circ g_2$, then $[\![t_1]\!]^{\mathrm{T}} = [\![t_2]\!]^{\mathrm{T}}$ and $g_1 \leq g_2$.

First,

$$\begin{array}{ll}
\multicolumn{2}{l}{[\![t_1]\!]^{\mathrm{T}} \circ \eta^{\mathrm{T}} \circ g_1} \\
= & (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{id}, [\![t_1]\!] \circ ! \rangle) \circ \langle \mathsf{id}, \mathsf{e} \circ ! \rangle \circ g_1 \\
= & \langle g_1, \mathsf{m} \circ \langle \mathsf{e}, [\![t_1]\!] \rangle \circ !_{\mathbf{1}} \rangle \\
= & \langle g_1, [\![t_1]\!] \rangle
\end{array}$$

Similarly, $[\![t_2]\!]^{\mathrm{T}} \circ \eta^{\mathrm{T}} \circ g_2 = \langle g_2, [\![t_1]\!] \rangle$. Therefore we know immediately that $[\![t_1]\!] \leq [\![t_2]\!]$ and $g_1 \leq g_2$. Because we assumed that the set $\{[\![t]\!] \mid t \in T\}$ is discretely ordered, $[\![t_1]\!] = [\![t_2]\!]$. Therefore $[\![t_1]\!]^{\mathrm{T}} = [\![t_2]\!]^{\mathrm{T}}$ as well.

Prop. #6: For all objects X, $\mathsf{reduce}_X \circ [\![t]\!]_{L\mathbf{T}X}^{\mathrm{T}} \circ \eta_{L\mathbf{T}X}^{\mathrm{T}} = L[\![t]\!]_X^{\mathrm{T}}$.

$$\begin{tikzcd}
LA \arrow[d, "\langle \mathsf{id}, f \circ ! \rangle"'] & & A \arrow[ll, "\mathsf{up}"'] \arrow[rr, "\mathsf{up}"] \arrow[d, "\langle \mathsf{id}, f \circ ! \rangle"] & & LA \arrow[ddd, dashed, "L\langle \mathsf{id}, f \circ ! \rangle"'] & & \mathbf{1} \arrow[ll, "\perp"'] \arrow[rr, "\perp"] \arrow[d, "\langle \mathsf{id}, f \circ ! \rangle"] \arrow[dddll, "\perp"'] & & LA \arrow[d, "\langle \mathsf{id}, f \circ ! \rangle"] \\
LA \times B \arrow[ddrr, "\mathsf{id} \times \mathsf{up}"'] & & A \times B \arrow[ll, "\mathsf{up} \times \mathsf{id}"'] \arrow[dd, "\mathsf{up} \times \mathsf{up}"] \arrow[ddrr, "\mathsf{up}"] & & & & \mathbf{1} \times B \arrow[rr, "\perp \times \mathsf{id}"] \arrow[dl, "!"] \arrow[dd, "\perp \times \mathsf{up}"] & & LA \times B \arrow[ddll, "\mathsf{id} \times \mathsf{up}"'] \\
& & & & & \mathbf{1} \arrow[dl, "\perp"] & & & \\
& & LA \times LB \arrow[rr, "\mathsf{smash}"'] & & L(A \times B) & & LA \times LB \arrow[ll, "\mathsf{smash}"] & &
\end{tikzcd}$$

Figure 3.14: Diagram for strictness property proof.

We prove this property in three stages. First, we need to show that for any $f : \mathbf{1} \to B$,

$$\mathsf{smash}_{A,B} \circ (\mathsf{id}_A \times \mathsf{up}_B) \circ \langle \mathsf{id}_{LA}, f \circ !_{LA} \rangle = L\langle \mathsf{id}_A, f \circ !_A \rangle$$

This equation is proven separately because it requires that we check by cases (up versus $\perp$). We do this with the diagram in Figure 3.14, remembering that $Ly = (\mathsf{up} \circ y)_{\perp}$.

Let $g = \mathsf{m} \circ \langle \mathsf{id}_{\mathbf{T}}, [\![t]\!] \circ !_{\mathbf{T}} \rangle$. Because $\langle \mathsf{id}, g \circ \mathsf{e} \circ ! \rangle$, we know that

$$\mathsf{id}^{\star} \circ \langle \mathsf{id}, g \circ \mathsf{e} \circ ! \rangle = \mathsf{id}^{\star} \circ [\![t]\!]^{\mathrm{T}} \circ \eta^{\mathrm{T}} = [\![t]\!]^{\mathrm{T}}$$

Lastly $\mathsf{reduce}_X \circ [\![t]\!]^{\mathrm{T}}_X \circ \eta^{\mathrm{T}}_X$ expands to

$$L\mathsf{id}^{\star} \circ \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up}) \circ \langle \mathsf{id}, g \circ \mathsf{e} \circ ! \rangle$$

and the diagram in Figure 3.15 shows that it equals $L[\![t]\!]^{\mathrm{T}}$.

Prop. #7: For all objects X, $\delta^{\mathrm{T}}_X \circ \eta^{\mathrm{T}}_X = \mathsf{id}_X$.

$$\delta^{\mathrm{T}} \circ \eta^{\mathrm{T}} = \pi_1 \circ \langle \mathsf{id}, \mathsf{e} \circ ! \rangle = \mathsf{id}$$

Prop. #8: For all $f : X \to AX'$, $\delta^{\mathrm{T}}_{X'} \circ f^{\star} = \delta^{\mathrm{T}}_{X'} \circ f \circ \delta^{\mathrm{T}}_X$.

$\pi_1 \circ \alpha^r = \pi_1 \circ \pi_1$, so

$$\begin{aligned}
\delta^{\mathrm{T}} \circ f^{\star} &= \pi_1 \circ (\mathsf{id} \times \mathsf{m}) \circ \alpha^r \circ (f \times \mathsf{id}) \\
&= \pi_1 \circ \pi_1 \circ (f \times \mathsf{id}) \\
&= \pi_1 \circ f \circ \pi_1 \\
&= \delta^{\mathrm{T}} \circ f \circ \delta^{\mathrm{T}}
\end{aligned}$$

$$
\begin{tikzcd}
L(X \times \mathbf{T}) \arrow[r, "\langle \mathsf{id}, g \circ \mathsf{e} \circ ! \rangle"] \arrow[d, "\mathsf{id}"'] & L(X \times \mathbf{T}) \times \mathbf{T} \arrow[r, "\mathsf{id} \times \mathsf{up}"] & L(X \times \mathbf{T}) \times L\mathbf{T} \arrow[d, "\mathsf{smash}"] \\
L(X \times \mathbf{T}) \arrow[rr, "L\langle \mathsf{id}, g \circ \mathsf{e} \circ ! \rangle"] \arrow[drr, "L[\![t]\!]^{\mathrm{T}}"'] & & L((X \times \mathbf{T}) \times \mathbf{T}) \arrow[d, "L\mathsf{id}^{\star}"] \\
& & L(X \times \mathbf{T})
\end{tikzcd}
$$

Figure 3.15: Proof of property #6

Prop. #9: For all $t \in T$ and all objects X, $\delta^{\mathrm{T}}_X \circ [\![t]\!]^{\mathrm{T}}_X = \delta^{\mathrm{T}}_X$.

$$\delta^{\mathrm{T}} \circ [\![t]\!]^{\mathrm{T}} = \pi_1 \circ (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{id}, [\![t]\!] \circ ! \rangle) = \pi_1 = \delta^{\mathrm{T}}$$

Prop. #10: For all objects X_1 and X_2 of $\mathcal{C}$, $\delta^{\mathrm{T}}_{X_1 \times X_2} \circ \psi^{\mathrm{T}}_{X_1, X_2} = \delta^{\mathrm{T}}_{X_1} \times \delta^{\mathrm{T}}_{X_2}$.

$\pi_1 \circ \phi = \pi_1 \times \pi_1$, therefore

$$
\begin{aligned}
\delta^{\mathrm{T}} \circ \psi^{\mathrm{T}} &= \pi_1 \circ (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \beta) \circ \phi \\
&= \pi_1 \times \pi_1 \\
&= \delta^{\mathrm{T}} \times \delta^{\mathrm{T}}
\end{aligned}
$$

Prop. #11: For all objects X of $\mathcal{C}$, $L\delta^{\mathrm{T}}_X \circ \mathsf{reduce}_X = L\delta^{\mathrm{T}}_X \circ \delta^{\mathrm{T}}_{L\mathbf{T}X}$.

From the properties described in Chapter 2 for smashed products (and depicted in Figure 2.13), we know that

$$L\pi_1 \circ \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up}) = \pi_1$$

Therefore

$$
\begin{aligned}
L\delta^{\mathrm{T}} \circ \mathsf{reduce} &= L\delta^{\mathrm{T}} \circ L(\mathsf{id})^{\star} \circ \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up}) \\
&= L\delta^{\mathrm{T}} \circ L\delta^{\mathrm{T}} \circ \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up}) \\
&= L\pi_1 \circ L\pi_1 \circ \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up}) \\
&= L\pi_1 \circ \pi_1 \\
&= L\delta^{\mathrm{T}} \circ \delta^{\mathrm{T}}
\end{aligned}
$$

To see that this arrow cost structure is strong, from [26] we know that $(\mathbf{T}, \eta^{\mathrm{T}}, (-)^{\star}, \tau^{\mathrm{T}})$ forms a strong monad, therefore we only need to show that reduce has the required properties.

First, we can simplify $\mathbf{T}f$ and τ^{T}. If $f : X \to Y$,

$$
\begin{aligned}
\mathbf{T}f &= (\eta^{\mathrm{T}} \circ f)^{\star} \\
&= (\mathsf{id} \times \mathsf{m}) \circ \alpha^r \circ (\langle \mathsf{id}, \mathsf{e} \circ ! \rangle \times \mathsf{id}) \circ (f \times \mathsf{id}) \\
&= (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{e} \circ !, \mathsf{id} \rangle) \circ (f \times \mathsf{id}) && \text{product isomorphism} \\
&= f \times \mathsf{id} && \text{monoid property}
\end{aligned}
$$

Also,

$$\begin{array}{rcll}\tau^{\mathrm{T}} & = & \psi^{\mathrm{T}} \circ (\mathsf{id} \times \eta^{\mathrm{T}}) & \\ & = & (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \beta) \circ \phi \circ (\mathsf{id} \times \langle \mathsf{id}, \mathsf{e} \circ ! \rangle) & \\ & = & (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \beta) \circ (\mathsf{id} \times \langle \mathsf{id}, \mathsf{e} \circ ! \rangle) \circ \sigma & \text{product isomorphism} \\ & = & (\mathsf{id} \times \mathsf{m}) \circ (\mathsf{id} \times \langle \mathsf{e} \circ !, \mathsf{id} \rangle) \circ \sigma & \text{product isomorphism} \\ & = & \sigma & \text{monoid property}\end{array}$$

where $\sigma = \alpha^l \circ (\mathsf{id} \times \beta) \circ \alpha^r$ is the product isomorphism such that $\sigma_{A,B,C} : (A \times B) \times C \to (A \times C) \times B$. Lastly, because $\tau^L = \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up})$, we can sometimes use the following expansion of reduce:

$$\mathsf{reduce}_X = L(\mathsf{id}_X \times \mathsf{m}) \circ L\alpha^r_{X,\mathbf{T},\mathbf{T}} \circ \tau^L_{X \times \mathbf{T}, TC}$$

We can now show that the extra properties of reduce (taken from both the definitions of a proper and strong arrow cost structure) hold. Because we know that we have a strong monad we will freely used mixed notation.

- For all morphisms $f : X \to \mathbf{T}X'$, $\mathsf{reduce}_{X'} \circ \mathbf{T}\mathsf{up}_{\mathbf{T}X'} \circ \mathbf{T}f = \mathsf{up}_{\mathbf{T}X'} \circ (f)^\star$

$$\begin{array}{lll}\mathsf{reduce} \circ \mathbf{T}\mathsf{up} \circ \mathbf{T}f & & \\ \quad = & L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up}) \circ (\mathsf{up} \times \mathsf{id}) \circ (f \times \mathsf{id}) & \\ \quad = & L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ \mathsf{smash} \circ (\mathsf{up} \times \mathsf{up}) \circ (f \times \mathsf{id}) & \\ \quad = & L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ \mathsf{up} \circ (f \times \mathsf{id}) & \text{property of } \mathsf{smash} \\ \quad = & \mathsf{up} \circ (\mathsf{id} \times \mathsf{m}) \circ \alpha^r \circ (f \times \mathsf{id}) & \text{naturality of } \mathsf{up} \\ \quad = & \mathsf{up} \circ (f)^\star & \end{array}$$

- For all morphisms $f : X \to L\mathbf{T}Y$, $\mathsf{reduce}_{X'} \circ \mathbf{T}(\mathsf{reduce}_{X'} \circ \mathbf{T}f)^\perp = (\mathsf{reduce}_{X'} \circ \mathbf{T}f)^\perp \circ \mathsf{reduce}_X$, First, note that

$$\begin{array}{lll}\mathsf{reduce} \circ \mathbf{T}(\mathsf{reduce} \circ \mathbf{T}f)^\perp & & \\ \quad = & L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ \tau^L \circ ((L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ \tau^L \circ \mathbf{T}f)^\perp \times \mathsf{id}) & \\ \quad = & L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ \tau^L \circ (L(\mathsf{id} \times \mathsf{m}) \times \mathsf{id}) \circ (L\alpha^r \times \mathsf{id}) \circ ((\tau^L \circ \mathbf{T}f)^\perp \times \mathsf{id}) & \\ & & \text{monad prop. of } (-)^\perp \\ \quad = & L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ L((\mathsf{id} \times \mathsf{m}) \times \mathsf{id}) \circ L(\alpha^r \times \mathsf{id}) \circ \tau^L \circ ((\tau^L \circ \mathbf{T}f)^\perp \times \mathsf{id}) & \\ & & \text{naturality of } \tau^L \\ \quad = & L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ L((\mathsf{id} \times \mathsf{m}) \times \mathsf{id}) \circ L(\alpha^r \times \mathsf{id}) & \\ & \qquad \circ (\tau^L \circ (\tau^L \times \mathsf{id}) \circ ((f \times \mathsf{id}) \times \mathsf{id}))^\perp \circ \tau^L & \\ & & \text{strength prop. of } \tau^L \\ \quad = & (L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ L((\mathsf{id} \times \mathsf{m}) \times \mathsf{id}) \circ L(\alpha^r \times \mathsf{id}) & \\ & \qquad \circ \tau^L \circ (\tau^L \times \mathsf{id}) \circ ((f \times \mathsf{id}) \times \mathsf{id}))^\perp \circ \tau^L & \\ & & \text{monad prop. of } (-)^\perp \end{array}$$

In figure 3.16, we see that

$$\begin{array}{ll} L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ L((\mathsf{id} \times \mathsf{m}) \times \mathsf{id}) \circ L(\alpha^r \times \mathsf{id}) \circ \tau^L \circ (\tau^L \times \mathsf{id}) \circ ((f \times \mathsf{id}) \times \mathsf{id}) & \\ \qquad = \; L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ \tau^L \circ (f \times \mathsf{id}) \circ (\mathsf{id} \times \mathsf{m}) \circ \alpha^r & \end{array}$$

$$
\begin{tikzcd}
(X \times \mathbf{T}) \times \mathbf{T} \arrow[r, "\alpha^r"] \arrow[dd, "(f \times \mathsf{id}) \times \mathsf{id}"] & X \times (\mathbf{T} \times \mathbf{T}) \arrow[r, "\mathsf{id} \times \mathsf{m}"] \arrow[dd, "f \times \mathsf{id}"] & X \times \mathbf{T} \arrow[dd, "f \times \mathsf{id}"] & \\
& & & \\
(L(X \times \mathbf{T}) \times \mathbf{T}) \times \mathbf{T} \arrow[r, "\alpha^r"] \arrow[d, "\tau^L \times \mathsf{id}"] & L(X \times \mathbf{T}) \times (\mathbf{T} \times \mathbf{T}) \arrow[r, "\mathsf{id} \times \mathsf{m}"] \arrow[dd, "\tau^L"] & L(X \times \mathbf{T}) \times \mathbf{T} \arrow[dd, "\tau^L"] & \\
L((X \times \mathbf{T}) \times \mathbf{T}) \times \mathbf{T} \arrow[d, "\tau^L"] & & & \\
L(((X \times \mathbf{T}) \times \mathbf{T}) \times \mathbf{T}) \arrow[r, "L\alpha^r"] \arrow[d, "L(\alpha^r \times \mathsf{id})"] & L((X \times \mathbf{T}) \times (\mathbf{T} \times \mathbf{T})) \arrow[r, "L(\mathsf{id} \times \mathsf{m})"] \arrow[dr, "L\alpha^r"] & L((X \times \mathbf{T}) \times \mathbf{T}) \arrow[ddr, "L\alpha^r"] & \\
L((X \times (\mathbf{T} \times \mathbf{T})) \times \mathbf{T}) \arrow[r, "L\alpha^r"] \arrow[dd, "L((\mathsf{id} \times \mathsf{m}) \times \mathsf{id})"] & L(X \times ((\mathbf{T} \times \mathbf{T}) \times \mathbf{T})) \arrow[r, "L(\mathsf{id} \times \alpha^l)"] \arrow[dd, "L(\mathsf{id} \times (\mathsf{m} \times \mathsf{id}))"] & L(X \times (\mathbf{T} \times (\mathbf{T} \times \mathbf{T}))) \arrow[dr, "L(\mathsf{id} \times (\mathsf{id} \times \mathsf{m}))"'] & \\
& & & L(X \times (\mathbf{T} \times \mathbf{T})) \arrow[dl, "L(\mathsf{id} \times \mathsf{m})"] \\
L((X \times \mathbf{T}) \times \mathbf{T}) \arrow[r, "L\alpha^r"'] & L(X \times (\mathbf{T} \times \mathbf{T})) \arrow[r, "L(\mathsf{id} \times \mathsf{m})"'] & L(X \times \mathbf{T}) &
\end{tikzcd}
$$

Figure 3.16: Part of the proof that the cost structure is strong

Aside from naturality and product isomorphisms, the diagram uses two strength properties of τ^L and one of the associative monoid properties. From this we then know that

$$
\begin{aligned}
&\mathsf{reduce} \circ \mathbf{T}(\mathsf{reduce} \circ \mathbf{T}f)^{\perp} \\
&\quad = (L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ \tau^L \circ (f \times \mathsf{id}) \circ (\mathsf{id} \times \mathsf{m}) \circ \alpha^r)^{\perp} \circ \tau^L \\
&\quad = (\mathsf{reduce} \circ (f \times \mathsf{id}) \circ (\mathsf{id} \times \mathsf{m}) \circ \alpha^r)^{\perp} \circ \tau^L \\
&\quad = (\mathsf{reduce} \circ (f \times \mathsf{id}))^{\perp} \circ L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^r \circ \tau^L \\
&\quad = (\mathsf{reduce} \circ \mathbf{T}f)^{\perp} \circ \mathsf{reduce}
\end{aligned}
$$

- $\mathsf{reduce}_{X_1 \times X_2} \circ \mathbf{T}L\tau^{\mathrm{T}}_{X_1,X_2} \circ \mathbf{T}\tau^{L}_{\mathbf{T}X_1,X_2} \circ \tau^{\mathrm{T}}_{L\mathbf{T}X_1,X_2} = L\tau^{\mathrm{T}}_{X_1,X_2} \circ \tau^{L}_{\mathbf{T}X_1,X_2} \circ (\mathsf{reduce}_{X_1} \times \mathsf{id}_{X_2})$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{reduce} \circ \mathbf{T}L\tau^{\mathrm{T}} \circ \mathbf{T}\tau^{L} \circ \tau^{\mathrm{T}}} \\
= & L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^{r} \circ \tau^{L} \circ (L\sigma \times \mathsf{id}) \circ (\tau^{L} \times \mathsf{id}) \circ \sigma & \\
= & L(\mathsf{id} \times \mathsf{m}) \circ L\alpha^{r} \circ L(\sigma \times \mathsf{id}) \circ \tau^{L} \circ (\tau^{L} \times \mathsf{id}) \circ \sigma & \text{naturality of } \tau^{L} \\
= & L(\mathsf{id} \times \mathsf{m}) \circ L\phi \circ L\alpha^{r} \circ \tau^{L} \circ (\tau^{L} \times \mathsf{id}) \circ \sigma & \text{product isomorphism} \\
= & L(\mathsf{id} \times \mathsf{m}) \circ L\phi \circ \tau^{L} \circ \alpha^{r} \circ \sigma & \text{strength prop. of } \tau^{L} \\
= & L(\mathsf{id} \times \mathsf{m}) \circ L\phi \circ \tau^{L} \circ (\mathsf{id} \times \beta) \circ \alpha^{r} & \text{product isomorphism} \\
= & L(\mathsf{id} \times \mathsf{m}) \circ L\phi \circ L(\mathsf{id} \times \beta) \circ \tau^{L} \circ \alpha^{r} & \text{naturality of } \tau^{L} \\
= & L(\mathsf{id} \times \mathsf{m}) \circ L\phi \circ L(\mathsf{id} \times \beta) \circ L\alpha^{r} \circ \tau^{L} \circ (\tau^{L} \times \mathsf{id}) & \text{strength prop. of } \tau^{L} \\
= & L(\mathsf{id} \times \mathsf{m}) \circ L\sigma \circ L(\alpha^{r} \times \mathsf{id}) \circ \tau^{L} \circ (\tau^{L} \times \mathsf{id}) & \text{product isomorphism} \\
= & L(\mathsf{id} \times \mathsf{m}) \circ L\sigma \circ \tau^{L} \circ (L\alpha^{r} \times \mathsf{id}) \circ (\tau^{L} \times \mathsf{id}) & \text{naturality of } \tau^{L} \\
= & L\sigma \circ L((\mathsf{id} \times \mathsf{m}) \times \mathsf{id}) \circ \tau^{L} \circ (L\alpha^{r} \times \mathsf{id}) \circ (\tau^{L} \times \mathsf{id}) & \text{naturality of } \alpha^{r} \\
= & L\sigma \circ \tau^{L} \circ (L(\mathsf{id} \times \mathsf{m}) \times \mathsf{id}) \circ (L\alpha^{r} \times \mathsf{id}) \circ (\tau^{L} \times \mathsf{id}) & \text{naturality of } \tau^{L} \\
= & L\tau^{\mathrm{T}} \circ \tau^{L} \circ (\mathsf{reduce} \times \mathsf{id}) &
\end{array}
$$

Thus $(\mathbf{T}, \eta^{\mathrm{T}}, (-)^{\star}, \psi^{\mathrm{T}}, [\![-]\!]^{\mathrm{T}}, \delta^{\mathrm{T}}, \mathsf{reduce})$ is strong. $\square$

3.6.4 Arrow cost structures and PDom

We have already shown that **PDom** has all the properties needed to be a base category for an arrow cost structure. Furthermore, the set T, discretely ordered, is a monoid object with a non-trivial $[\![-]\!]$ function. Therefore we can form a non-trivial arrow cost structure in **PDom**. In **PDom** the relevant functions and morphisms become the following:

$$
\begin{array}{rcl}
\mathbf{T}X & = & X \times \mathbf{T} \\
\eta^{\mathrm{T}}_{X}(x) & = & \langle x, 0 \rangle \\
(f)^{\star}(x,t) & = & \langle x', t' + t \rangle, \text{ where } f(x) = \langle x', t' \rangle \\
\psi^{\mathrm{T}}_{X_1,X_2}(\langle x_1, t_1 \rangle, \langle x_2, t_2 \rangle) & = & \langle \langle x_1, x_2 \rangle, t_2 + t_1 \rangle \\
[\![t]\!]^{\mathrm{T}}_{X}(x, t') & = & (x, t' + t) \\
\delta^{\mathrm{T}}_{X}(x,t) & = & x \\
\mathsf{reduce}_X(\bot, t) & = & \bot \\
\mathsf{reduce}_X(\mathsf{up}(x, t'), t) & = & \mathsf{up}(x, t' + t)
\end{array}
$$

In $\mathbf{PDom}^{\rightarrow}$, the morphisms become pairs of functions. To show each function we let int and ext be functions such that for any morphism (h, d) in $\mathbf{PDom}^{\rightarrow}$, $\mathsf{int}(h, d) = h$ and $\mathsf{ext}(h, d) = d$. Thus

$$
\begin{array}{rcl}
C\rho(\bot) & = & \bot \\
C\rho(\mathsf{up}(x,t)) & = & \mathsf{up}(\rho(x)) \\
\mathsf{int}(\eta_{\rho})(x) & = & \mathsf{up}\langle x, 0 \rangle \\
\mathsf{ext}(\eta_{\rho})(y) & = & \mathsf{up}(y)
\end{array}
$$

$$
\begin{aligned}
\mathsf{int}((h,d)^*)(\bot) &= \bot \\
\mathsf{int}((h,d)^*)(\mathsf{up}(x,t)) &= \begin{cases} \bot & h(x) = \bot \\ \mathsf{up}(x', t'+t) & h(x) = \mathsf{up}(x',t') \end{cases} \\
\mathsf{ext}((h,d)^*)(\bot) &= \bot \\
\mathsf{ext}((h,d)^*)(\mathsf{up}(y)) &= d(y) \\
\mathsf{int}(\llbracket t \rrbracket_\rho)(\bot) &= \bot \\
\mathsf{int}(\llbracket t \rrbracket_\rho)(\mathsf{up}(x,t')) &= \mathsf{up}(x, t'+t) \\
\mathsf{ext}(\llbracket t \rrbracket_\rho)(y) &= y \\
\mathsf{int}(\psi_{\rho_1,\rho_2})\langle \bot, z\rangle &= \bot \\
\mathsf{int}(\psi_{\rho_1,\rho_2})\langle z, \bot\rangle &= \bot \\
\mathsf{int}(\psi_{\rho_1,\rho_2})\langle \mathsf{up}(x_1,t_1), \mathsf{up}(x_2,t_2)\rangle &= \mathsf{up}(\langle x_1, x_2\rangle, t_2+t_1) \\
\mathsf{ext}(\psi_{\rho_1,\rho_2})\langle \bot, y\rangle &= \bot \\
\mathsf{ext}(\psi_{\rho_1,\rho_2})\langle y, \bot\rangle &= \bot \\
\mathsf{ext}(\psi_{\rho_1,\rho_2})\langle \mathsf{up}(y_1), \mathsf{up}(y_2)\rangle &= \mathsf{up}(y_1, y_2)
\end{aligned}
$$

3.6.5 The intensional semantics in the arrow category

In specifying the intensional semantics using the category we gain an additional advantage: we can automatically convert the extensional types into intensional types. For cost structures in general, we assume that for each ground type g there exists an object A_g^{V} such that EA_g^{V} is A_g, and we assume that for each constructed type δ of arity n there exists an n-ary functor F_δ^{V} such that $EF_\delta^{\mathrm{V}} = F_\delta$. For the arrow category, however, we can set A_g^{V} to $\mathsf{id}_{A_g} : A_g \to A_g$ and we can set F_δ^{V} to the standard extended functor of F_δ, i.e., for all morphisms $p_i : X_i \to D_i$, $1 \le i \le n$ in $\mathcal{C}$,

$$F_\delta^{\mathrm{V}}(p_1, \ldots, p_n) = F_\delta(p_1, \ldots p_n)$$

and for morphisms $h_i : X_i \to X_i'$, $d_i : D_i \to D_i'$, $1 \le i \le n$,

$$F_\delta^{\mathrm{V}}((h_1, d_1), \ldots, (h_n, d_n)) = (F_\delta(h_1, \ldots, h_1), F_\delta(d_1, \ldots, d_n))$$

We still need to specify the intensional constants as they contain cost information we cannot derive from the extensional semantics; however, the building blocks for the constants frequently have obvious extensions in the arrow category.

3.7 The intensional semantics for FL

In chapter 2 we defined specific constants for the language FL, gave extensional meanings for them (both operationally and denotationally), and showed that their meanings were adequate. In this section we define their intensional semantics, both operational and denotational, and show that they satisfy the assumptions needed for intensional soundness. Intensional adequacy and extensional soundness then follow from the properties of the cost structure.

The intensional operational semantics for FL are listed in Figure 3.17. The costs for each constants is chosen primarily to be simple yet still show some interesting examples. Thus for most constants, the cost is constant. To show some other types of costs, for integer equality and

$$\text{apply}\,(+, \overline{n}, \overline{m}) \stackrel{t_+}{\Rightarrow} \overline{n+m} \qquad \text{apply}\,(\times, \overline{n}, \overline{m}) \stackrel{\mathsf{t}_*(n,m)}{\Rightarrow} \overline{n*m}$$

$$\text{apply}\,(\dot{-}, \overline{n}, \overline{m}) \stackrel{t_{\dot{-}}}{\Rightarrow} \overline{0} \quad (n \leq m) \qquad \text{apply}\,(\dot{-}, \overline{n}, \overline{m}) \stackrel{t_{\dot{-}}}{\Rightarrow} \overline{n-m} \quad (n > m)$$

$$\text{apply}\,(=, \overline{n}, \overline{n}) \stackrel{t_=}{\Rightarrow} \texttt{true} \qquad \text{apply}\,(=, \overline{n}, \overline{m}) \stackrel{t_{\neq}}{\Rightarrow} \texttt{false} \quad (n \neq m)$$

$$\text{apply}\,(\leq, \overline{n}, \overline{m}) \stackrel{t_\leq}{\Rightarrow} \texttt{true} \quad (n < m) \qquad \text{apply}\,(\leq, \overline{n}, \overline{m}) \stackrel{t_\leq}{\Rightarrow} \texttt{false} \quad (n > m)$$

$$\text{apply}\,(\texttt{fst}, <v_1, v_2>) \stackrel{t_{\texttt{fst}}}{\Rightarrow} v_1 \qquad \text{apply}\,(\texttt{snd}, <v_1, v_2>) \stackrel{t_{\texttt{snd}}}{\Rightarrow} v_2$$

$$\frac{v_1(v) \stackrel{t}{\Rightarrow}_{\mathrm{v}} v'}{\text{apply}\,(\texttt{case}, \texttt{inl}(v), v_1, v_2) \stackrel{t+t_{\texttt{case}}}{\Rightarrow} v'} \qquad \frac{v_2(v) \stackrel{t}{\Rightarrow}_{\mathrm{v}} v'}{\text{apply}\,(\texttt{case}, \texttt{inr}(v), v_1, v_2) \stackrel{t+t_{\texttt{case}}}{\Rightarrow} v'}$$

$$\text{apply}\,(\texttt{head}, v_1{::}v_2) \stackrel{t_{\texttt{head}}}{\Rightarrow} v_1 \qquad \text{apply}\,(\texttt{tail}, v_1{::}v_2) \stackrel{t_{\texttt{tail}}}{\Rightarrow} v_2$$

Figure 3.17: Call-by-value intensional operational semantics for FL

multiplication, the cost is dependent on the input. For the equality test the cost differs depending on whether or not the input is 0 (i.e., a successful test has a different cost than an unsuccessful test). For multiplication, the cost is dependent on the input: $\mathsf{t}_*(n, m)$ is an (unspecified) function from integers to the set of costs (for example, $\mathsf{t}_*(n, m)$ could be $(\log(n)+\log(m))t_{\text{mult}}$). This models cases where multiplication is not constant. Thus we can examine cases where the intensional semantics is not a trivial extension of the extensional semantics.

For the extensional semantics, we assumed that the category in use was **PDom**. Similarly we are free to limit the cost structure to any consistent with **PDom**. In particular, we assume that our cost structure is derived from the arrow cost structure as shown in section 3.6.4. This enables us to specify morphisms on costs simply by writing functions, which is particularly useful for complicated cost functions like the one used for multiplication.

We already know how to convert the objects into $\mathbf{PDom}^{\rightarrow}$. Converting many of the morphisms is also straightforward. Let the superscript ^{I} be added to any of the type-specific morphisms (like plus or inl) to refer to the versions in $\mathbf{PDom}^{\rightarrow}$. In most cases, the intensional semantics is simply the extensional semantics with the type-specific morphisms replaced by their intensional versions, the lifting monad morphisms replaced by the relevant cost structure morphisms, and then cost added. When the cost of an operation is not constant the intensional semantics may become more complicated. In all cases, however, applying E to an intensional meanings returns its extensional counterpart.

3.7.1 Natural numbers

As **nat** is a ground type, $\mathbf{N}^I = \mathsf{id}_{\mathbf{N}}$. Furthermore, for all $m \in \mathbf{N}$, ${\mathsf{n}_m}^I = (\mathsf{n}_m, \mathsf{n}_m)$, etc. The cost of multiplication is a function of its input, so we need a method to convert the values into costs. Let mcost be a function such that $\mathsf{mcost}(n, m) = [\![\mathsf{t}_*(n, m)]\!]$. As $\mathbf{N} \times \mathbf{N}$ is discretely ordered, all functions from it are continuous, therefore mcost is also a morphism from $\mathbf{N}\times\mathbf{N}$ to $\mathbf{T}$ that calculates the cost associated with elements in $\mathbf{N}\times\mathbf{N}$. To create a version of mcost in the intensional category, let $\mathsf{imcost} : \mathbf{N}^I \times \mathbf{N}^I \to C(\mathbf{N}^I \times \mathbf{N}^I)$ be $(\mathsf{up}_{\mathbf{T}(\mathbf{N}\times\mathbf{N})} \circ \langle \mathsf{id}_{\mathbf{N}\times\mathbf{N}}, \mathsf{mcost}\rangle, \mathsf{up}_{\mathbf{N}\times\mathbf{N}})$. The intensional part of the morphism, $\mathsf{up}_{\mathbf{T}(\mathbf{N}\times\mathbf{N})} \circ \langle \mathsf{id}_{\mathbf{N}\times\mathbf{N}}, \mathsf{mcost}\rangle$ takes two integers, and adds cost corresponding

$$
\begin{array}{rcl}
\mathcal{C}^{\mathrm{V}}[\![\overline{n}]\!] & = & \eta \circ \mathsf{n}_n{}^I \\
\mathcal{C}^{\mathrm{V}}[\![=]\!] & = & \mathsf{cond} \circ \langle \mathsf{id}, \langle [\![t_=]\!] \circ \eta, [\![t_{\neq}]\!] \circ \eta \rangle\rangle \circ \mathsf{eq}^I \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}[\![\leq]\!] & = & [\![t_{\leq}]\!] \circ \eta \circ \mathsf{leq}^I \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}[\![+]\!] & = & [\![t_+]\!] \circ \eta \circ \mathsf{plus}^I \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}[\![\dot{-}]\!] & = & [\![t_-]\!] \circ \eta \circ \mathsf{minus}^I \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}[\![\times]\!] & = & (\eta \circ \mathsf{times}^I)^* \circ \mathsf{imcost} \circ (\pi_2 \times \mathsf{id})
\end{array}
$$

Figure 3.18: Intensional meanings of the integer constants

$$
\begin{array}{rcl}
\mathcal{C}^{\mathrm{V}}[\![\mathtt{pair}]\!] & = & \eta \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}[\![\mathtt{fst}]\!] & = & [\![t_{\mathrm{fst}}]\!] \circ \eta \circ \pi_1 \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}[\![\mathtt{snd}]\!] & = & [\![t_{\mathrm{snd}}]\!] \circ \eta \circ \pi_2 \circ \pi_2
\end{array}
$$

Figure 3.19: Intensional meanings of the product constants

(via mcost) to the input, lifting the final result. Thus imcost adds non-constant cost to an item, and the non-constant cost corresponds to mcost. In particular,

$$\mathsf{imcost} \circ \langle \mathsf{n}_n{}^I, \mathsf{n}_m{}^I \rangle = [\![\mathsf{mcost}(n, m)]\!] \circ \eta \circ \langle \mathsf{n}_n{}^I, \mathsf{n}_m{}^I \rangle$$

Because $L\delta^{\mathrm{T}} \circ \mathsf{up} \circ \langle \mathsf{id}, \mathsf{mcost} \rangle = \mathsf{up}$, imcost is a valid morphism in $\mathbf{PDom}^I$.

The only other complicated case is the equality test. For this constant we take advantage of knowing that the cost can be computed from the result of the function as well as from the input, thus we can derive the result from the cost structure morphisms and cond.

The intensional semantics are listed in Figure 3.18. Note that

$$\mathcal{C}^{\mathrm{V}}[\![+]\!] \circ \langle\rangle(\mathsf{n}_n{}^I, \mathsf{n}_m{}^I) = [\![t_+]\!] \circ \eta \circ \mathsf{n}_{n+m}{}^I$$

and

$$\mathcal{C}^{\mathrm{V}}[\![\times]\!] \circ \langle\rangle(\mathsf{n}_n{}^I, \mathsf{n}_m{}^I) = [\![\mathsf{mcost} n, m]\!] \circ \eta \circ \mathsf{n}_{n*m}{}^I$$

as expected. Furthermore, given that $\mathsf{cond} \circ \langle \mathsf{id}, \langle \mathsf{up}, \mathsf{up} \rangle\rangle = \mathsf{up}$ in $\mathbf{PDom}$ and that $E\mathsf{imcost} = \mathsf{up}$, it is clear that for all of the integer constants c, $E\mathcal{C}^{\mathrm{V}}[\![c]\!] = \mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c]\!]$.

3.7.2 Products

We already know, from the conditions of cost structures, that $\mathbf{PDom}^{\rightarrow}$ has products and that E preserves them. Because the cost of taking the first or second element of a pair is independent of the elements of the pair, we can derive the meaning of the constants directly from their extensional meaning. The intensional meanings are listed in Figure 3.19.

$$
\begin{aligned}
\mathcal{C}^{\mathrm{V}}[\![\texttt{inl}]\!] &= \eta \circ \mathsf{inl} \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}[\![\texttt{inr}]\!] &= \eta \circ \mathsf{inr} \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}[\![\texttt{case}]\!] &= [\![t_{\mathrm{case}}]\!] \circ \mathsf{case}(\mathsf{vleft}, \mathsf{vright}) \circ (\pi_2 \times \mathsf{id}) \circ \alpha^r \\
\mathsf{vleft} &= \mathsf{app} \circ \beta \circ (\mathsf{id} \times \pi_1) \\
\mathsf{vright} &= \mathsf{app} \circ \beta \circ (\mathsf{id} \times \pi_2)
\end{aligned}
$$

Figure 3.20: Intensional meanings of the sum constants

3.7.3 Sums

The category $\mathbf{PDom}^{\rightarrow}$ has coproducts, and E preserves them. There are two possible definitions for the intensional version of $\mathsf{case}(-,-)$, i.e.,

$$\mathsf{case}((h_1, d_1), (h_2, d_2))^I = \mathsf{uncurry}([\mathsf{curry}(h_1, d), \mathsf{curry}(h_2, d_2)])$$

or

$$\mathsf{case}((h_1, d_1), (h_2, d_2))^I = (\mathsf{case}(h_1, h_2), \mathsf{case}(d_1, d_2))$$

We need not, however, choose between them because, by the properties of the pullback used to define the exponential, the two definitions are equivalent. Therefore, the intensional meanings of the sum constants will resemble the extensional meanings except for the additional cost information, which is constant for `case` (and 0 for the two constructors). For simplicity, we write $\mathsf{case}(-,-)^I$ as $\mathsf{case}(-,-)$ when there is no chance of confusion between the intensional and extensional versions. Figure 3.20 lists the intensional semantics.

3.7.4 Lists

The list functor is raised to the arrow category in the same manner as products and coproducts were lifted, i.e., for $p : X \to D$, $\mathsf{List}^I(p) = \mathsf{List}\,(p) : \mathsf{List}\,(X) \to \mathsf{List}\,(D)$. We can convert the list morphisms similarly, i.e. $\mathsf{nil}_p^I = (\mathsf{nil}_X, \mathsf{nil}_D)$ and so forth. For hd and tl, however, we do not simply want the lifted versions; $\mathsf{hd} : \mathsf{List}\,(A) \to LA$, so $\mathsf{hd}^I : \mathsf{List}^I(p) \to Lp$. For our semantics, however, we would prefer to use a morphism from $\mathsf{List}^I(p)$ to Cp. Therefore let $\mathsf{ihd}_p : \mathsf{List}^I(p) \to Cp$ be $(L\mathsf{zero} \circ \mathsf{hd}_X, \mathsf{hd}_D)$, which is similar to $\mathsf{List}^I p$ except that it also adds 0 cost, making it more useful for the semantics. Similarly, let $\mathsf{itl}_p : \mathsf{List}^I(p) \to C\mathsf{List}^I(p)$ be $(L\mathsf{zero} \circ \mathsf{tl}_X, \mathsf{tl}_D)$. The properties between the list constants, listed in Figure 3.21, are therefore similar, but not identical, to the ones found in Figure 2.18 of Chapter 2, in that they use a cost structure instead of the lifting monad.

We now derive the intensional meanings for the constants by using the special morphisms ihd and itl, converting the lifting monad, and adding cost. The intensional semantics are listed in Figure 3.22. Note that $\mathcal{C}^{\mathrm{V}}[\![\texttt{head}]\!] \circ \langle\rangle(\mathsf{nil}^I) = \bot$ and $\mathcal{C}^{\mathrm{V}}[\![\texttt{head}]\!] \circ \langle\rangle(\mathsf{cons}^I) \circ \langle p_1, p_2\rangle = [\![t_{\mathrm{head}}]\!] \circ \eta \circ p_1$ as expected.

For the rest of this chapter we will use single symbols (such as f) for morphisms in the arrow category unless we need to access the individual morphisms within the pair. This allows clearer formulas without the loss of any critical information.

$$
\begin{array}{ll}
\mathsf{ihd}_p \circ \mathsf{nil}_p^I = \bot_{Tp} & \mathsf{ihd}_p \circ \mathsf{cons}_p^I = \eta_p \circ \pi_1 \\
\mathsf{itl}_p \circ \mathsf{nil}_p^I = \bot_{Tp} & \mathsf{itl}_p \circ \mathsf{cons}_p^I = \eta_p \circ \pi_2 \\
\mathsf{ihd}_{p'} \circ \mathsf{List}^I(h,d) = (\eta \circ (h,d))^* \circ \mathsf{ihd}_p & \mathsf{itl}_{p'} \circ \mathsf{List}^I(h,d) = (\eta \circ \mathsf{List}^I(h,d))^* \circ \mathsf{itl}_p \\
\mathsf{null?}_p^I \circ \mathsf{nil}_p^I = \mathsf{tt} & \mathsf{null?}_p^I \circ \mathsf{cons}_p^I = \mathsf{ff} \circ ! \\
\mathsf{null?}_{p'}^I \circ \mathsf{List}^I(h,d) = \mathsf{null?}_p & \mathsf{List}^I(h,d) \circ \mathsf{nil}_p^I = \mathsf{nil}_{p'}^I \\
\mathsf{List}^I(h,d) \circ \mathsf{cons}_p^I = \mathsf{cons}_{p'}^I \circ ((h,d) \times (\eta \circ \mathsf{List}^I(h,d))^*) &
\end{array}
$$

Figure 3.21: List properties in the arrow category

$$
\begin{array}{rcl}
\mathcal{C}^{\mathrm{V}}[\![\mathtt{nil}]\!] & = & \eta \circ \mathsf{nil}^I \\
\mathcal{C}^{\mathrm{V}}[\![\mathtt{cons}]\!] & = & \eta \circ \mathsf{cons}^I \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}[\![\mathtt{head}]\!] & = & [\![t_{\mathrm{head}}]\!] \circ \mathsf{ihd} \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}[\![\mathtt{tail}]\!] & = & [\![t_{\mathrm{tail}}]\!] \circ \mathsf{itl} \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}[\![\mathtt{nil?}]\!] & = & [\![t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathsf{null?}^I \circ \pi_2
\end{array}
$$

Figure 3.22: Intensional meanings of the list constants

3.7.5 Soundness of the intensional semantics

For soundness, all of the constructors and constants of 0 arity factor through η therefore are sound. Therefore, by Lemma 3.4.8, the meaning of all values factor through η. For all of the other constants outside of `case`, there are no premises the the rules of their operational semantics, therefore such a constant c is sound if for any rule of the form

$$\mathrm{vapply}(c, v_1, \ldots, v_{n-1}, v_n) \overset{t}{\Rightarrow} v$$

where n is the arity of c, if $c \in \mathbf{Const}_{\tau_1 \to \tau_n \to \tau}$ and for each $1 \le i \le n$, v_i is a closed variable of type τ_i, then

$$\mathcal{C}^{\mathrm{V}}[\![c]\!] \circ \langle\rangle(f_1, \ldots, f_n) = [\![t]\!] \circ \mathcal{V}[\![v : \tau]\!]$$

where for each $1 \le i \le n$, $\mathcal{V}[\![v_i : \tau_i]\!] = \eta \circ f_i$.

The proof that the constants are sound is mostly straightforward; we prove three of the more difficult cases here.

Multiplication: $\mathrm{apply}(\times\ \overline{n}, \overline{m}) \overset{\mathsf{t}_*(n,m)}{\Rightarrow} \overline{n * m}$:

First note that for any $n \in \mathbf{N}$,

$$
\begin{array}{rcl}
\mathcal{V}[\![\overline{n} : \mathbf{nat}]\!] & = & \mathcal{C}^{\mathrm{V}}[\![\overline{n}]\!] \circ !_1 \\
& = & \eta \circ {\mathsf{n}_n}^I
\end{array}
$$

Also, as $\langle \mathsf{id}, \mathsf{mcost}\rangle \circ \langle \mathsf{n}_n, \mathsf{n}_m\rangle = [\![\mathsf{t}_*(n,m)]\!]^* \circ \mathsf{zero} \circ \langle \mathsf{n}_n, \mathsf{n}_m\rangle$,

$$
\begin{array}{rcl}
\mathsf{imcost} \circ \langle {\mathsf{n}_n}^I, {\mathsf{n}_m}^I\rangle & = & (\mathsf{up} \circ \langle \mathsf{id}, \mathsf{mcost}\rangle, \mathsf{up}) \circ (\langle \mathsf{n}_n, \mathsf{n}_m\rangle, \langle \mathsf{n}_n, \mathsf{n}_m\rangle) \\
& = & (\mathsf{up} \circ \langle \mathsf{id}, \mathsf{mcost}\rangle \circ \langle \mathsf{n}_n, \mathsf{n}_m\rangle, \mathsf{up} \circ \langle \mathsf{n}_n, \mathsf{n}_m\rangle) \\
& = & (\mathsf{up} \circ [\![\mathsf{t}_*(n,m)]\!]^* \circ \mathsf{zero} \circ \langle \mathsf{n}_n, \mathsf{n}_m\rangle, \mathsf{up} \circ \langle \mathsf{n}_n, \mathsf{n}_m\rangle) \\
& = & (L[\![\mathsf{t}_*(n,m)]\!]^*, \mathsf{id}) \circ (\mathsf{up} \circ \mathsf{zero}, \mathsf{up}) \circ \langle {\mathsf{n}_n}^I, {\mathsf{n}_m}^I\rangle \\
& = & [\![\mathsf{t}_*(n,m)]\!] \circ \eta \circ \langle {\mathsf{n}_n}^I, {\mathsf{n}_m}^I\rangle
\end{array}
$$

Therefore

$$\begin{aligned}
&\mathcal{C}^{\mathrm{V}}[\![\times]\!] \circ \langle\rangle(\mathsf{n}_n{}^I, \mathsf{n}_m{}^I) \\
&\quad = (\eta \circ \mathsf{times}^I)^* \circ \mathsf{imcost} \circ (\pi_2 \times \mathsf{id}) \circ \langle\rangle(\mathsf{n}_n{}^I, \mathsf{n}_m{}^I) \\
&\quad = (\eta \circ \mathsf{times}^I)^* \circ \mathsf{imcost} \circ \langle \mathsf{n}_n{}^I, \mathsf{n}_m{}^I\rangle \\
&\quad = (\eta \circ \mathsf{times}^I)^* \circ [\![\mathsf{t}_*(n,m)]\!] \circ \eta \circ \langle \mathsf{n}_n{}^I, \mathsf{n}_m{}^I\rangle \\
&\quad = [\![\mathsf{t}_*(n,m)]\!] \circ \eta \circ \mathsf{times}^I \circ \langle \mathsf{n}_n{}^I, \mathsf{n}_m{}^I\rangle \\
&\quad = [\![\mathsf{t}_*(n,m)]\!] \circ \eta \circ \mathsf{n}_{m*n}{}^I \\
&\quad = [\![\mathsf{t}_*(n,m)]\!] \circ \mathcal{V}[\![\overline{n*m} : \mathbf{nat}]\!]
\end{aligned}$$

Equality test: $\mathrm{apply}(=\overline{n}, \overline{n}) \overset{t_=}{\Rightarrow} \mathtt{true}$:

Again, $\mathcal{V}[\![\overline{n} : \mathbf{nat}]\!] = \eta \circ \mathsf{n}_n{}^I$, so

$$\begin{aligned}
&\mathcal{C}^{\mathrm{V}}[\![=]\!] \circ \langle\rangle(\mathsf{n}_n{}^I, \mathsf{n}_n{}^I) \\
&\quad = \mathsf{cond} \circ \langle \mathsf{id}, \langle [\![t_=]\!] \circ \eta, [\![t_{\neq}]\!] \circ \eta\rangle\rangle \circ \mathsf{eq}^I \circ (\pi_2 \times \mathsf{id}) \circ \langle\rangle(\mathsf{n}_n{}^I, \mathsf{n}_n{}^I) \\
&\quad = \mathsf{cond} \circ \langle \mathsf{id}, \langle [\![t_=]\!] \circ \eta, [\![t_{\neq}]\!] \circ \eta\rangle\rangle \circ \mathsf{eq}^I \circ \langle \mathsf{n}_n{}^I, \mathsf{n}_n{}^I\rangle \\
&\quad = \mathsf{cond} \circ \langle \mathsf{id}, \langle [\![t_=]\!] \circ \eta, [\![t_{\neq}]\!] \circ \eta\rangle\rangle \circ \mathsf{tt} \\
&\quad = \mathsf{cond} \circ \langle \mathsf{tt}, \langle [\![t_=]\!] \circ \eta \circ \mathsf{tt}, \langle [\![t_{\neq}]\!] \circ \eta \circ \mathsf{tt}\rangle\rangle \\
&\quad = [\![t_=]\!] \circ \eta \circ \mathsf{tt} \\
&\quad = [\![t_=]\!] \circ \mathcal{V}[\![\mathtt{true} : \mathbf{bool}]\!]
\end{aligned}$$

Left sum case: $\dfrac{v_1(v) \overset{t}{\Rightarrow}_{\mathrm{v}} v'}{\mathrm{apply}(\mathtt{case}, \mathtt{inl}(v), v_1, v_2) \overset{t+t_{\mathrm{case}}}{\Rightarrow} v'}$:

By assumption $\mathtt{case}\ \mathtt{inl}(v)\ \mathtt{of}\ \mathtt{left} : v_1\ \mathtt{right} : v_2$ is well typed, therefore there exist types τ, τ_1, and τ_2 such that $\vdash v : \tau_1$, $\vdash v_1 : \tau_1 \to \tau$, and $\vdash v_2 : \tau_2 \to \tau$. Furthermore, v, v_1 and v_2 are all values, so there exist $y : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau_1]\!]$, $y_1 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau_1 \to \tau]\!]$, and $y_2 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau_2 \to \tau]\!]$ such that $\mathcal{V}[\![v : \tau_1]\!] = \eta \circ y$, $\mathcal{V}[\![v_1 : \tau_1 \to \tau]\!] = \eta \circ y_1$, $\mathcal{V}[\![v_2 : \tau_2 \to \tau]\!] = \eta \circ y_2$, and

$$\mathcal{V}[\![\mathtt{inl}(v) : \tau_1 + \tau_2]\!] = \eta \circ \mathsf{inl} \circ \pi_2 \circ y = \eta \circ \mathsf{inl} \circ y$$

By the definition of soundness for constants, we can assume that $\mathcal{V}[\![v_1(v) : \tau]\!] = [\![t]\!] \circ \mathcal{V}[\![v' : \tau]\!]$, thus

$$\begin{aligned}
\mathsf{vleft} \circ \langle y, \langle y_1, y_2\rangle\rangle &= \mathsf{app} \circ \langle y_1, y\rangle \\
&= \mathsf{app}^* \circ \psi \circ \langle \eta \circ y_1, \eta \circ y\rangle \\
&= \mathcal{V}[\![v_1(v) : \tau]\!] \\
&= [\![t]\!] \circ \mathcal{V}[\![v' : \tau]\!]
\end{aligned}$$

Therefore

$$\begin{aligned}
&\mathcal{V}[\![\mathtt{case}\ \mathtt{inl}(v)\ \mathtt{of}\ \mathtt{left} : v_1\ \mathtt{right} : v_2 : \tau]\!] \\
&\quad = [\![t_{\mathrm{case}}]\!] \circ \mathsf{case}(\mathsf{vleft}, \mathsf{vright}) \circ (\pi_2 \times \mathsf{id}) \circ \alpha^r \circ \langle\rangle(\mathsf{inl} \circ y, y_1, y_2) \\
&\quad = [\![t_{\mathrm{case}}]\!] \circ \mathsf{case}(\mathsf{vleft}, \mathsf{vright}) \circ \langle \mathsf{inl} \circ y, \langle y_1, y_2\rangle\rangle \\
&\quad = [\![t_{\mathrm{case}}]\!] \circ \mathsf{vleft} \circ \langle y, \langle y_1, y_2\rangle\rangle \\
&\quad = [\![t_{\mathrm{case}}]\!] \circ [\![t]\!] \circ \mathcal{V}[\![v' : \tau]\!] \\
&\quad = [\![t + t_{\mathrm{case}}]\!] \circ \mathcal{V}[\![v' : \tau]\!]
\end{aligned}$$

3.8 Examples

In this section we examine a number of small example programs. With these programs we not only demonstrate how the denotational semantics analyzes the particular examples, but also how we can, through the use of abbreviations, simplify the structure of the meanings so that calculating costs is as simple denotationally as it was operationally, plus we have a compositional framework making it easier to build up complicated calculations from simpler ones.

There is significant mathematical structure embedded in the denotational semantics. Some of the structure is superficial: the form of the semantic clause for an application,

$$\mathcal{V}[\![\Gamma \vdash e_1(e_2) : \tau]\!] = \mathsf{app}^* \circ \psi \circ \langle \mathcal{V}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!], \mathcal{V}[\![\Gamma \vdash e_2 : \tau']\!]\rangle$$

is complex primarily because we express it in terms of monad operations and app. In this section we will simply write the above as

$$\mathbf{capply}(\mathcal{V}[\![\Gamma \vdash e_1 : \to \tau]\!], \mathcal{V}[\![\Gamma \vdash e_2 : \tau']\!])$$

(where "c" stands for "cost"), and determine those properties that will allow us to reduce equations. Thus while in the rest of this chapter we were primarily concerned with using familiar structures, making it easier to prove general properties, in this section we are primarily concerned with compact structures, making it easier to interpret the meaning of a particular example.

We also include significant structure in the form of the meanings themselves: The meaning $\mathcal{V}[\![\Gamma \vdash e : \tau]\!]$ is "actually" a pair of morphisms in an arrow category (at least for FL). We only needed this category, however, to prove separability and adequacy. When examining particular examples we only need the general properties of a Cartesian closed category with lifts, so we can safely treat $\mathcal{V}[\![\Gamma \vdash e : \tau]\!]$ as a single morphism and not be concerned about its internal structure.

The extra structure that ensures adequacy gives us an additional advantage: We can handle cases where an expression fails to terminate. For example, Lemma 3.4.7 was restricted to values because application of constants depends on assumptions about the structure of its input. Without adequacy we do not have sufficient knowledge about the general structure of non-terminating inputs to enable us to predict their actions. With adequacy we can be certain that the meaning of a closed expression of type τ is either $\perp$ or of the form $[\![t]\!] \circ \eta \circ y$ where $y : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ and where $\eta \circ y$ is the meaning of some value v. With the substitution lemma we can also know that a similar property holds for arbitrary expressions when combined with suitable environments.

3.8.1 Abbreviating applications

Let **capply** be the family of functions such that for $f : A \to C\mathcal{T}^{\mathrm{V}}[\![\tau_1 \to \ldots \to \tau_n \to \tau]\!]$ and for $f_i : A \to C\mathcal{T}^{\mathrm{V}}[\![\tau_i]\!]$, $1 \le i \le n$, $\mathbf{capply}(f) = f$ and

$$\mathbf{capply}(f, f_1, \ldots, f_n) = \mathsf{app}^* \circ \psi \circ \langle \mathbf{capply}(f, f_1, \ldots, f_{n-1}), f_n\rangle$$

Then if $\Gamma \vdash e : \tau_1 \to \ldots \to \tau_n \to \tau$, and $\Gamma \vdash e_i : \tau_i$, $1 \le i \le n$,

$$\mathcal{V}[\![\Gamma \vdash ee_1 \ldots e_n : \tau]\!] = \mathbf{capply}(\mathcal{V}[\![\Gamma \vdash e : \tau_0]\!], \mathcal{V}[\![\Gamma \vdash e_1 : \tau_1]\!], \ldots, \mathcal{V}[\![\Gamma \vdash e_n : \tau_n]\!])$$

where $\tau_0 = \tau_1 \to \ldots \to \tau_n \to \tau$. This form is particularly useful when each item is not a simple expression. We can also now succinctly state formally the property that all functions are strict: For all $f, f_1, \ldots, f_n$

$$\mathbf{capply}(f, f_1, \ldots f_n) = \perp$$

whenever any of the f_i's (or f) are $\perp$. This property can be proven easily by induction on $n - i$. The reverse property, of course, is not necessarily true: $\mathbf{capply}(\mathcal{V}[\![\texttt{head}]\!], \mathcal{V}[\![\texttt{nil}]\!]) = \perp$ even though neither component is $\perp$.

What happens if none of the f_i's are $\perp$? Then each f_i (if their domains are all $\mathbf{1}$) can be safely assumed to have the form $[\![t_i]\!] \circ \eta \circ y_i$ for some $y_i : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau_i]\!]$. If we assume that all the f_i's and f are derived from expressions, then we can also assume that $\mathbf{capply}(f, f_1, \ldots, f_i)$ is either $\perp$ or of the form $[\![t]\!] \circ \eta \circ y$ for $1 \leq i \leq n$ and for some appropriate y.

If we make these assumptions, and if we further assume that the monoid of costs is commutative, then we can show, using the structure of the definitions of **capply**, that

$$\mathbf{capply}(f, f_1, \ldots, f_n) = [\![t_1 + \ldots + t_n]\!] \circ \mathbf{capply}(f, \eta \circ y_1, \ldots, \eta \circ y_n)$$

For this to hold it is critical that costs must be commutative: If individual $\mathbf{capply}(f, f_1, \ldots, f_i)$ have non-zero costs (which is true for abstractions) then those costs would be distributed between the t_i's. Because we are primarily interested in the total cost and not their order of accumulation, we will normally be interested in a commutative notion of cost, so this requirement does not adversely affect our understanding of individual programs.

The property just listed indicates that the costs of the inputs to an argument affect the cost of the output in a highly regular manner and thus can be ignored when analyzing particular programs. Therefore let **apply** be a similar family of functions such that $\mathbf{apply}(f) = f$ and

$$\mathbf{apply}(f, g_1, \ldots, g_n) = \mathsf{app}^* \circ \psi \circ \langle \mathbf{apply}(f, g_1, \ldots, g_{n-1}), \eta \circ g_n \rangle$$

These functions use values for the arguments rather than computations. It is clear from the definition that $\mathbf{apply}(f, g_1, \ldots, g_n) = \mathbf{capply}(f, \eta \circ g_1, \ldots, \eta \circ g_n)$.

There are some additional properties that remove the **capply** or **apply** constructs altogether in certain cases. From Lemma 3.4.7 it is clear that

$$\mathbf{apply}(\mathcal{V}[\![\Gamma \vdash c : \tau]\!], g_1, \ldots, g_n) = \mathcal{C}^{\mathrm{V}}[\![c]\!] \circ \langle\rangle(g_1, \ldots, g_n)$$

when n is the arity of c. We can derive a similar result for expressions consisting of abstractions:

Theorem 3.8.1 *Suppose that* $f = [\![t]\!] \circ \mathcal{V}[\![\Gamma \vdash \texttt{lam x}_1 \ldots \texttt{lam x}_n.e : \tau_0]\!] \circ \langle\rangle(f_1, \ldots, f_k)$*, where* τ_0 *equals* $\tau_1 \to \ldots \to \tau_n \to \tau$ *(i.e., each* x_i *has type* τ_i*), and* $\langle\rangle(f_1, \ldots, f_n) : X \to \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]$*. Furthermore suppose that for* $1 \leq i \leq n$*,* $g_i : X \to \mathcal{T}^{\mathrm{V}}[\![\tau_i]\!]$*. Then*

$$\begin{aligned}
&\mathbf{apply}(f, g_1, \ldots, g_i)\\
&\quad = [\![it_{\mathrm{app}} + t]\!] \circ \mathcal{V}[\![\Gamma, \texttt{x}_1 : \tau_1, \ldots, \texttt{x}_\texttt{i} : \tau_i \vdash \texttt{lam x}_{\texttt{i+1}} \ldots \texttt{lam x}_\texttt{n}.e : \tau_{i+1} \to \ldots \to \tau_n \to \tau]\!]\\
&\qquad\qquad \circ \langle\rangle(f_1, \ldots, f_k, \eta \circ g_1, \ldots, \eta \circ g_i)
\end{aligned}$$

We included the case where f was the meaning of an expression that requires a non-trivial amount of calculation before becoming an abstraction because we also want to handle expressions such as

$$\texttt{rec z.lam x}_1 \ldots \texttt{lam x}_\texttt{n}.e$$

If $f = \mathcal{V}[\![\Gamma \vdash \texttt{rec z}.e : \tau]\!]$, then

$$\begin{aligned}
f &= \mathsf{fixp}([\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, \mathsf{z} : \tau \vdash e : \tau]\!])\\
&= [\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, \mathsf{z} : \tau \vdash e : \tau]\!] \circ \langle \mathsf{id}, \mathsf{fixp}([\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, \mathsf{z} : \tau \vdash e : \tau]\!]) \rangle\\
&= [\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, \mathsf{z} : \tau \vdash e : \tau]\!] \circ \langle\rangle(f)
\end{aligned}$$

Therefore, for expressions headed by recursion, we obtain the following:

Corollary 3.8.2 *If* $f = \mathcal{V}[\![\Gamma \vdash \texttt{rec z.lam x}_1\texttt{...lam x}_\texttt{n}.e : \tau_0]\!]$, *where* $\tau_0 = \tau_1 \to \ldots \to \tau_n \to \tau$, *then for all* $g_i : \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!] \to \mathcal{T}^{\mathrm{V}}[\![\tau_i]\!]$, $1 \leq i \leq n$,

$$\begin{array}{ll}\multicolumn{2}{l}{\mathbf{apply}(f, g_1, \ldots, g_i)} \\ = & [\![it_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, \texttt{z} : \tau, \texttt{x}_1 : \tau_1, \ldots \texttt{x}_\texttt{i} : \tau_i \vdash \texttt{lam x}_{\texttt{i+1}}\texttt{...lam x}_\texttt{n}.e : \tau']\!] \\ & \circ \langle\rangle(f, \eta \circ g_1, \ldots, \eta \circ g_i)\end{array}$$

where $\tau' = \tau_{i+1} \to \ldots \to \tau_n \to \tau$.

Lastly, if $\Gamma \vdash e : \tau$, Γ' is a type environment with n variables, none of which are in Γ, and $\langle\rangle(f_1, \ldots, f_n) : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\Gamma']\!]$, then, by repeated applications of the Drop Lemma,

$$\mathcal{V}[\![\Gamma, \Gamma' \vdash e : \tau]\!] \circ \langle\rangle(f_1, \ldots, f_n) = \mathcal{V}[\![\Gamma \vdash e : \tau]\!]$$

This property can be used inside **capply** under the same assumptions on Γ and Γ':

$$\begin{array}{ll}\multicolumn{2}{l}{\mathbf{capply}(\mathcal{V}[\![\Gamma, \Gamma' \vdash e : \tau]\!], g_1, \ldots, g_k) \circ \langle\rangle(f_1, \ldots, f_n)} \\ = & \mathbf{capply}(\mathcal{V}[\![\Gamma, \Gamma' \vdash e : \tau]\!] \circ \langle\rangle(f_1, \ldots, f_n), g_1 \circ \langle\rangle(f_1, \ldots, f_n), \ldots, g_k \circ \langle\rangle(f_1, \ldots, f_n)) \\ = & \mathbf{capply}(\mathcal{V}[\![\Gamma \vdash e : \tau]\!], g_1 \circ \langle\rangle(f_1, \ldots, f_n), \ldots, g_k \circ \langle\rangle(f_1, \ldots, f_n))\end{array}$$

Therefore the behavior of programs and constants depend only on the behavior of their free variables, as we would expect from compositionality.

3.8.2 Abbreviating conditionals

We will also find it useful to form abbreviations for conditionals. Let

$$\mathbf{cond}(f, g, h) = \mathsf{cond}^* \circ \psi \circ \langle f, \eta \circ \langle g, h\rangle\rangle$$

We may also display this in multiple lines, as

$$\begin{array}{ll}\multicolumn{2}{l}{\mathbf{cond}(f)\ \mathbf{of}} \\ \quad\mathbf{True:} & g \\ \quad\mathbf{False:} & h\end{array}$$

Either way, f is a morphism from X to $C\mathbf{B}$, for some object X, while g and h are morphisms from X to X' for some object X'. Thus $\mathbf{cond}(f, g, h)$ is a morphism from X to X'.

At times we may want to state the test as a general boolean expression (such as $n = 0$) rather than a morphism. Let **bool** be a function such that

$$\mathbf{bool}(x) = \begin{cases} \mathsf{tt} \circ ! & x \text{ is true} \\ \mathsf{ff} \circ ! & x \text{ if false}\end{cases}$$

Then let $\mathbf{cond}(\{x\}, g, h) = \mathbf{cond}(\eta \circ \mathbf{bool}(x), g, h)$. The braces around x are to emphasize that x is not a morphism but a boolean expression.

There are a number of simple properties associated with these functions. For example,

$$\mathbf{cond}(\{x\}, g, h) = \mathsf{cond}^* \circ \psi \circ \langle \eta \circ \mathbf{bool}(x), \eta \circ \langle g, h\rangle\rangle = \mathsf{cond} \circ \langle \mathbf{bool}(x), \langle g, h\rangle\rangle$$

Thus if x is true, then $\mathbf{cond}(\{x\}, g, h) = g$; if x is false, then $\mathbf{cond}(\{x\}, g, h) = h$. Additionally we have the following properties:

- For any morphisms $f : X \to C\mathbf{B}$, $g, h : X \to X'$, and $k : Y \to X$,

$$\mathbf{cond}(f, g, h) \circ k = \mathbf{cond}(f \circ k, g \circ k, h \circ k)$$

- For any morphisms $x : X \to \mathbf{B}$, $g, h : X \to CX'$, and any cost t,

$$\mathbf{cond}([\![t]\!] \circ \eta \circ x, g, h) = [\![t]\!] \circ \mathbf{cond}(\eta \circ x, g, h)$$

- For any boolean condition x, any morphisms $g, h : X \to CX'$, and any cost t,

$$\mathbf{cond}([\![t]\!] \circ \eta \circ \mathbf{bool}(x), g, h) = [\![t]\!] \circ \mathbf{cond}(\{x\}, g, h)$$

- If $\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau$ for type τ, type environment Γ, and expressions e_1, e_2, e_3, then

$$\begin{aligned} &\mathcal{V}[\![\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau]\!] \\ &\quad = \quad \mathbf{cond}(\mathcal{V}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!], [\![t_{\text{true}}]\!] \circ \mathcal{V}[\![\Gamma \vdash e_2 : \tau]\!], [\![t_{\text{false}}]\!] \circ \mathcal{V}[\![\Gamma \vdash e_3 : \tau]\!]) \end{aligned}$$

3.8.3 Properties of free variables

Because both **capply** and **cond** distribute with composition on the right, we can apply a few properties to variables as well. From the semantic meaning of a variable, we know that

$$\mathcal{V}[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_i : \tau_i]\!] \circ \langle\rangle(f_1, \ldots, f_n) = f_i$$

and we can use this fact within applications or conditionals. For example

$$\begin{aligned} &\mathcal{V}[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e(x_i) : \tau]\!] \circ \langle\rangle(f_1, \ldots, f_n) \\ &\quad = \quad \mathbf{capply}(\mathcal{V}[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau_i \to \tau]\!], \circ\langle\rangle(f_1, \ldots, f_n), f_i) \end{aligned}$$

or simply $\mathbf{capply}(\mathcal{V}[\![e : \tau_i \to \tau]\!], f_i)$ when e is closed. Similarly, for any constant c of type $\tau \to \mathbf{bool}$,

$$\begin{aligned} &\mathcal{V}[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \texttt{if } c(x_i) \texttt{ then } e_1 \texttt{ else } e_2 : \tau]\!] \circ \langle\rangle(f_1, \ldots, f_n) \\ &\quad = \quad \mathbf{cond}(\mathbf{capply}(\mathcal{V}[\![c]\!], f_i)) \ \mathbf{of} \\ &\qquad\qquad \mathbf{True:} \quad \mathcal{V}[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_1 : \tau]\!] \circ \langle\rangle(f_1, \ldots, f_n) \\ &\qquad\qquad \mathbf{False:} \quad \mathcal{V}[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_2 : \tau]\!] \circ \langle\rangle(f_1, \ldots, f_n) \end{aligned}$$

3.8.4 The `length` program

We first examine the program

$$\texttt{length} = \texttt{rec len.lam l.if nil?(l) then } \overline{0} \texttt{ else } \overline{1} + \texttt{len(tail(l))}$$

which is the program examined operationally in Section 3.2.1. The analysis of this program is clearer if we include some abbreviations for lists. Let $[] = \mathsf{nil}^I$ and let

$$[a_1, \ldots, a_n] = \mathsf{cons}^I \circ \langle a_1, [a_2, \ldots, a_n]\rangle$$

$$\begin{aligned}
\mathcal{C}^{\mathrm{V}}[\![\texttt{head}]\!] \circ \langle\rangle([a_1, \ldots, a_n]) &= [\![t_{\mathrm{head}}]\!] \circ \mathbf{cond}(\{n = 0\}, \bot, \eta \circ a_1)\\
\mathcal{C}^{\mathrm{V}}[\![\texttt{tail}]\!] \circ \langle\rangle([a_1, \ldots, a_n]) &= [\![t_{\mathrm{tail}}]\!] \circ \mathbf{cond}(\{n = 0\}, \bot, \eta \circ [a_2, \ldots, a_n])\\
\mathcal{C}^{\mathrm{V}}[\![\texttt{nil?}]\!] \circ \langle\rangle([a_1, \ldots, a_n]) &= [\![t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathbf{bool}(n = 0)
\end{aligned}$$

Figure 3.23: FL list properties

where, for some type τ, a_1, ..., a_n are elements of $\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$; that is, they are morphisms from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$. If v is a closed value of type $\mathbf{list}(\tau)$, then there exists an n and a set of morphisms $a_1, \ldots, a_n : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ such that $\mathcal{V}[\![v : \mathbf{list}(\tau)]\!] = [a_1, \ldots, a_n]$. Therefore this notation covers all the lists of interest.

We now can derive properties with lists, as denoted above, and the FL constants themselves, based on the properties of the primitive list morphisms listed in Figure 3.21. These new properties are listed in Figure 3.23. They do not include any properties related to List^I because those have no direct equivalent in FL. The property concerning `nil?` is designed particularly for use with the conditional: from it we know that, for any morphisms $f_1, f_2 : \mathbf{1} \to C\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$,

$$\mathbf{cond}(\mathcal{C}^{\mathrm{V}}[\![\texttt{nil?}]\!] \circ \langle\rangle([a_1, \ldots, a_n]), f_1, f_2) = [\![t_{\mathrm{nil?}}]\!] \circ \mathbf{cond}(\{n = 0\}, f_1, f_2)$$

We now have sufficient tools to calculate costs for `length` without exposing internal complex calculations. Suppose that $[a_1, \ldots, a_n] : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{list}(\tau)]\!]$, where n may be 0. Let τ_0 be the type $\mathbf{list}(\tau) \to \mathbf{nat}$. We determine the semantics by first calculating some of the inner parts of the expression. Thus, we begin by calculating the meaning of `len(tail(l))`, with an environment $r = \langle\rangle(z, \eta \circ y)$:

$$\begin{aligned}
&\mathcal{V}[\![\texttt{len} : \tau_0, \texttt{l} : \mathbf{list}(\tau) \vdash \texttt{len(tail(l))} : \mathbf{nat}]\!] \circ r\\
&\quad = \mathbf{capply}(z, \mathcal{V}[\![\texttt{len} : \tau_0, \texttt{l} : \mathbf{list}(\tau) \vdash \texttt{tail(l)} : \mathbf{list}(\tau)]\!] \circ r)\\
&\quad = \mathbf{capply}(z, \mathbf{capply}(\mathcal{V}[\![\texttt{tail}]\!], \eta \circ y))\\
&\quad = \mathbf{capply}(z, \mathcal{C}^{\mathrm{V}}[\![\texttt{tail}]\!] \circ \langle\rangle(y))
\end{aligned}$$

We next calculate the meaning of $\overline{1} + \texttt{len(tail(l))}$:

$$\begin{aligned}
&\mathcal{V}[\![\texttt{len} : \tau_0, \texttt{l} : \mathbf{list}(\tau) \vdash \overline{1} + \texttt{len(tail(l))} : \mathbf{nat}]\!] \circ r\\
&\quad = \mathbf{capply}(\mathcal{V}[\![+]\!], \mathcal{V}[\![\overline{1}]\!], \mathbf{capply}(z, \mathcal{C}^{\mathrm{V}}[\![\texttt{tail}]\!] \circ \langle\rangle(y)))
\end{aligned}$$

Lastly,

$$\begin{aligned}
&\mathcal{V}[\![\texttt{len} : \tau_0, \texttt{l} : \mathbf{list}(\tau) \vdash \texttt{nil?(l)} : \mathbf{bool}]\!] \circ r\\
&\quad = \mathcal{C}^{\mathrm{V}}[\![\texttt{nil?}]\!] \circ \langle\rangle(y)
\end{aligned}$$

which becomes $[\![t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathbf{bool}(n = 0)$ when $y = \eta \circ [a_1, \ldots, a_n]$.

Combining the calculations,

$$\begin{aligned}
&\mathbf{apply}(\mathcal{V}[\![\texttt{length} : \tau_0]\!], [a_1, \ldots, a_n])\\
&\quad = [\![t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\texttt{len} : \tau_0, \texttt{l} : \mathbf{list}(\tau) \vdash \texttt{if nil?(l) then } \overline{0} \texttt{ else } \overline{1} + \texttt{len(tail(l))} : \mathbf{nat}]\!]\\
&\qquad\qquad \circ \langle\rangle(\mathcal{V}[\![\texttt{length} : \tau_0]\!], \eta \circ [a_1, \ldots, a_n])\\
&\quad = [\![t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \ \mathbf{cond}(\{n = 0\}) \ \mathbf{of}\\
&\qquad\qquad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \mathcal{V}[\![\overline{0}]\!]\\
&\qquad\qquad \mathbf{False:} \quad [\![t_{\mathrm{false}}]\!] \circ \mathbf{capply}(\mathcal{V}[\![+]\!], \mathcal{V}[\![\overline{1}]\!],\\
&\qquad\qquad\qquad \mathbf{capply}(\mathcal{V}[\![\texttt{length} : \tau_0]\!], \mathcal{C}^{\mathrm{V}}[\![\texttt{tail}]\!] \circ \langle\rangle([a_1, \ldots, a_n])))
\end{aligned}$$

We can now examine particular cases. If $y = []$, then $n = 0$, so

$$\begin{aligned}
&\mathbf{apply}(\mathcal{V}[\![\texttt{length} : \tau_0]\!], [])\\
&\quad = \quad [\![t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{n_0}^I\\
&\quad = \quad [\![t_{\mathrm{true}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{n_0}^I
\end{aligned}$$

If $y = [a_1, \ldots, a_n]$, where $n > 0$, then

$$\begin{aligned}
&\mathbf{apply}(\mathcal{V}[\![\texttt{length} : \tau_0]\!], [a_1, \ldots, a_n])\\
&\quad = \quad [\![t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ [\![t_{\mathrm{false}}]\!]\\
&\qquad\qquad \circ\, \mathbf{capply}(\mathcal{V}[\![+]\!], \mathcal{V}[\![\overline{1}]\!], \mathbf{capply}(\mathcal{V}[\![\texttt{length} : \tau_0]\!], [\![t_{\mathrm{tail}}]\!] \circ \eta \circ [a_2, \ldots, a_n]))\\
&\quad = \quad [\![t_{\mathrm{false}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!]\\
&\qquad\qquad \circ\, \mathbf{capply}(\mathcal{V}[\![+]\!], \mathcal{V}[\![\overline{1}]\!], [\![t_{\mathrm{tail}}]\!] \circ \mathbf{apply}(\mathcal{V}[\![\texttt{length} : \tau_0]\!], [a_2, \ldots, a_n]))
\end{aligned}$$

If we further assume that $\mathbf{apply}(\mathcal{V}[\![\texttt{length} : \tau_0]\!], [a_2, \ldots, a_n]) = [\![t]\!] \circ \eta \circ y'$ for some y' (i.e., if we assume that the evaluation terminates), then

$$\begin{aligned}
&\mathbf{apply}(\mathcal{V}[\![\texttt{length} : \tau_0]\!], [a_1, \ldots, a_n])\\
&\quad = \quad [\![t_{\mathrm{false}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathbf{capply}(\mathcal{V}[\![+]\!], \eta \circ \mathsf{n_1}^I, [\![t_{\mathrm{tail}}]\!] \circ [\![t]\!] \circ \eta \circ y')\\
&\quad = \quad [\![t + t_{\mathrm{tail}} + t_{\mathrm{false}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathbf{apply}(\mathcal{V}[\![+]\!], \mathsf{n_1}^I, y')\\
&\quad = \quad [\![t + t_{\mathrm{tail}} + t_{\mathrm{false}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ [\![t_+]\!] \circ \eta \circ \mathsf{plus}^I \circ \langle \mathsf{n_1}^I, y' \rangle\\
&\quad = \quad [\![t + t_+ + t_{\mathrm{tail}} + t_{\mathrm{false}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{plus}^I \circ \langle \mathsf{n_1}^I, y' \rangle
\end{aligned}$$

As it is straightforward, by induction on the length of the list, to show that `length` always terminates, we can write the cost function for `length` as follows:

$$\begin{aligned}
T(0) &= \quad t_{\mathrm{nil?}} + t_{\mathrm{true}} + t_{\mathrm{app}} + t_{\mathrm{rec}}\\
T(n) &= \quad T(n-1) + t_+ + t_{\mathrm{tail}} + t_{\mathrm{false}} + t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}
\end{aligned}$$

Using standard methods for solving recurrence equations, we can show that T is the following function:

$$T(n) = n(t_+ + t_{\mathrm{tail}} + t_{\mathrm{false}}) + (n+1)(t_{\mathrm{nil?}} + t_{\mathrm{app}} + t_{\mathrm{rec}}) + t_{\mathrm{true}}$$

which is the same function found operationally.

3.8.5 The Fibonacci function

The well-known Fibonacci series, 1, 1, 2, 3, 5, ..., is an interesting example for complexity analysis because the simplest algorithm for it is extremely inefficient. In this section we will analyze a program using the inefficient algorithm, and a more efficient one.

We first examine an inefficient program:

$$\texttt{fib1} = \texttt{rec f.lam n.if n} \leq \overline{2}\ \texttt{then}\ \overline{1}\ \texttt{else f(n}\dot{-}\overline{1}) + \texttt{f(n}\dot{-}\overline{2})$$

To find its meaning, let us first look at $\texttt{n}\dot{-}\overline{m}$ for some integer m. Because all values of type **nat** have form $\overline{k}$ for some k, we can safely assume that if $x : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{nat}]\!]$, then $x = \mathsf{n}_k{}^I$ for some integer k. Therefore we can assume that the element in the environment for n in the expression

$n \dot{-} \overline{m}$ is of the form $\eta \circ \mathsf{n}_n{}^I$. Let $\Gamma = \mathtt{f} : \mathbf{nat} \to \mathbf{nat}, \mathtt{n} : \mathbf{nat}$, and let $z_f : \mathbf{1} \to C\mathcal{T}^{\mathrm{V}}[\![\mathbf{nat} \to \mathbf{nat}]\!]$. Then

$$\begin{aligned} &\mathcal{V}[\![\Gamma \vdash \mathtt{n} \dot{-} \overline{m} : \mathbf{nat}]\!] \circ \langle\rangle(z_f, \eta \circ \mathsf{n}_n{}^I) \\ &\quad = \mathbf{capply}(\mathcal{V}[\![\dot{-}]\!], \eta \circ \mathsf{n}_n{}^I, \eta \circ \mathsf{n}_m{}^I) \\ &\quad = [\![t_-]\!] \circ \eta \circ \mathsf{minus}^I \circ \langle \mathsf{n}_n{}^I, \mathsf{n}_m{}^I \rangle \\ &\quad = \begin{cases} [\![t_-]\!] \circ \eta \circ \mathsf{n}_{n-m}{}^I & n \geq m \\ [\![t_-]\!] \circ \eta \circ \mathsf{n}_0{}^I & n < m \end{cases} \\ &\quad = [\![t_-]\!] \circ \eta \circ \mathsf{n}_{n \dot{-} m}{}^I \end{aligned}$$

Similarly,

$$\begin{aligned} &\mathcal{V}[\![\Gamma \vdash \mathtt{n} \leq \overline{2} : \mathbf{bool}]\!] \circ \langle\rangle(z_f, \eta \circ \mathsf{n}_n{}^I) \\ &\quad = \begin{cases} [\![t_\leq]\!] \circ \eta \circ \mathsf{tt} \circ ! & n \leq 2 \\ [\![t_\leq]\!] \circ \eta \circ \mathsf{ff} \circ ! & n > 2 \end{cases} \\ &\quad = [\![t_\leq]\!] \circ \eta \circ \mathbf{bool}(n \leq 2) \end{aligned}$$

Next we find that

$$\begin{aligned} &\mathcal{V}[\![\Gamma \vdash \mathtt{f}(\mathtt{n} \dot{-} \overline{1}) + \mathtt{f}(\mathtt{n} \dot{-} \overline{2}) : \mathbf{nat}]\!] \circ \langle\rangle(z_f, \eta \circ \mathsf{n}_n{}^I) \\ &\quad = \mathbf{capply}(\mathcal{V}[\![+]\!], \mathbf{capply}(z_f, [\![t_-]\!] \circ \eta \circ \mathsf{n}_{n \dot{-} 1}{}^I), \mathbf{capply}(z_f, [\![t_-]\!] \circ \eta \circ \mathsf{n}_{n \dot{-} 2}{}^I)) \\ &\quad = \mathbf{capply}(\mathcal{V}[\![+]\!], [\![t_-]\!] \circ \mathbf{apply}(z_f, \mathsf{n}_{n \dot{-} 1}{}^I), [\![t_-]\!] \circ \mathbf{apply}(z_f, \mathsf{n}_{n \dot{-} 2}{}^I)) \end{aligned}$$

Therefore, for the entire function, if we let

$$\mathsf{fb}(n) = \mathbf{apply}(\mathcal{V}[\![\mathtt{fib}]\!], \mathsf{n}_n{}^I)$$

then

$$\begin{aligned} \mathsf{fb}(n) &= \mathbf{capply}(\mathcal{V}[\![\mathtt{fib1}]\!], \mathsf{n}_n{}^I) \\ &= [\![t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \ \mathbf{cond}([\![t_\leq]\!] \circ \eta \circ \mathbf{bool}(n \leq 2)) \ \mathbf{of} \\ &\qquad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{n}_1{}^I \\ &\qquad \mathbf{False:} \quad [\![t_{\mathrm{false}}]\!] \circ \mathbf{capply}(\mathcal{V}[\![+]\!], [\![t_-]\!] \circ \mathsf{fb}(n-1), [\![t_-]\!] \circ \mathsf{fb}(n-2)) \\ &= [\![t_\leq + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \ \mathbf{cond}(\{n \leq 2\}) \ \mathbf{of} \\ &\qquad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{n}_1{}^I \\ &\qquad \mathbf{False:} \quad [\![t_{\mathrm{false}}]\!] \circ \mathbf{capply}(\mathcal{V}[\![+]\!], [\![t_-]\!] \circ \mathsf{fb}(n-1), [\![t_-]\!] \circ \mathsf{fb}(n-2)) \end{aligned}$$

If $n <= 2$, then

$$\mathsf{fb}(n) = [\![t_{\mathrm{true}} + t_\leq + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{n}_1{}^I$$

and if $n > 2$, then

$$\mathsf{fb}(n) = [\![t_{\mathrm{false}} + t_\leq + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathbf{capply}(\mathcal{V}[\![+]\!], [\![t_-]\!] \circ \mathsf{fb}(n-1), [\![t_-]\!] \circ \mathsf{fb}(n-2))$$

From these equations we can show by induction on n that for all n, $\mathsf{fb}(n)$ is not $\bot$. Therefore there exist functions fb_t and fb_v such that for all n, $\mathsf{fb}(n) = [\![\mathsf{fb}_t(n)]\!] \circ \eta \circ \mathsf{n}_{\mathrm{fb}_v(n)}{}^I$. For $n = 1$ or $n = 2$, $\mathsf{fb}_t(n) = t_{\mathrm{true}} + t_\leq + t_{\mathrm{app}} + t_{\mathrm{rec}}$ and $\mathsf{fb}_v(n) = 1$; for $n > 2$, we know that

$$\begin{aligned} \mathsf{fb}(n) &= [\![t_{\mathrm{false}} + t_\leq + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \\ &\qquad \circ \mathbf{capply}(\mathcal{V}[\![+]\!], [\![t_-]\!] \circ [\![\mathsf{fb}_t(n-1)]\!] \circ \eta \circ \mathsf{n}_{\mathrm{fb}_v(n-1)}{}^I, [\![t_-]\!] \circ [\![\mathsf{fb}_t(n-2)]\!] \circ \eta \circ \mathsf{n}_{\mathrm{fb}_v(n-2)}{}^I) \\ &= [\![t_{\mathrm{false}} + 2t_- + t_\leq + t_{\mathrm{app}} + t_{\mathrm{rec}} + \mathsf{fb}_t(n-1) + \mathsf{fb}_t(n-2)]\!] \circ \mathcal{C}^{\mathrm{V}}[\![+]\!] \\ &\qquad \circ \langle\rangle(\mathsf{n}_{\mathrm{fb}_v(n-1)}{}^I, \mathsf{n}_{\mathrm{fb}_v(n-2)}{}^I) \\ &= [\![t_+ + t_{\mathrm{false}} + 2t_- + t_\leq + t_{\mathrm{app}} + t_{\mathrm{rec}} + \mathsf{fb}_t(n-1) + \mathsf{fb}_t(n-2)]\!] \circ \mathsf{n}_{\mathrm{fb}_v(n-1) + \mathrm{fb}_v(n-2)}{}^I \quad (*) \end{aligned}$$

From (*) we can see that for $n > 2$, $\mathsf{fb}_v(n) = \mathsf{fb}_v(n-1) + \mathsf{fb}_v(n-2)$, so fb_v is, as expected, the Fibonacci function. From (*) we also see that the cost of the function, fb_t satisfies the following:

$$\begin{aligned}
\mathsf{fb}_t(1) = \mathsf{fb}_t(2) &= t_{\text{true}} + t_{\leq} + t_{\text{app}} + t_{\text{rec}} \\
\mathsf{fb}_t(n) &= t_{+} + t_{\text{false}} + 2t_{-} + t_{\leq} + t_{\text{app}} + t_{\text{rec}} + \mathsf{fb}_t(n-1) + \mathsf{fb}_t(n-2) \quad (n > 2)
\end{aligned}$$

Thus $\mathsf{fb}_t(n)$ is bounded below by t_{app} times the Fibonacci function. This demonstrates that the particular program is highly inefficient. The problem is that it recalculates the lower values of fb_t multiple times by calculating $\mathsf{fb}_t(n-1)$ and $\mathsf{fb}_t(n-2)$ independently.

A more efficient program uses an internal recursive function to accumulate the result from the onset, instead of repeatedly calculating subproblems. Let `ifib` and `fib2` be the following programs:

$$\begin{aligned}
\texttt{ifib} &= \texttt{rec f.lam a.lam b.lam n.if n} \leq \overline{1} \texttt{ then a else ifib b } (\texttt{a} + \texttt{b}) \ (\texttt{n} \dot{-} \overline{1}) \\
\texttt{fib2} &= \texttt{ifib } \overline{1}\ \overline{1}
\end{aligned}$$

The meaning of `ifib` is easily calculated: for $n = 1$ and any integers a and b,

$$\mathbf{apply}(\mathcal{V}[\![\texttt{ifib}]\!], \mathsf{n}_a{}^I, \mathsf{n}_b{}^I, \mathsf{n}_1{}^I) = [\![3t_{\text{app}} + t_{\leq} + t_{\text{rec}} + t_{\text{true}}]\!] \circ \eta \circ \mathsf{n}_a{}^I$$

while for $n > 1$,

$$\begin{aligned}
&\mathbf{apply}(\mathcal{V}[\![\texttt{ifib}]\!], \mathsf{n}_a{}^I, \mathsf{n}_b{}^I, \mathsf{n}_n{}^I) \\
&\qquad = [\![3t_{\text{app}} + t_{\leq} + t_{\text{rec}} + t_{\text{false}} + t_{-} + t_{+}]\!] \circ \mathbf{apply}(\mathcal{V}[\![\texttt{ifib}]\!], \mathsf{n}_b{}^I, \mathsf{n}_{a+b}{}^I, \mathsf{n}_{n-1}{}^I)
\end{aligned}$$

Therefore if $\mathbf{apply}(\mathcal{V}[\![\texttt{ifib}]\!], \mathsf{n}_a{}^I, \mathsf{n}_b{}^I, \mathsf{n}_n{}^I)$ is $[\![\mathsf{cst}(a,b,n)]\!] \circ \eta \circ \mathsf{val}(a,b,n)$, then we know that

$$\begin{aligned}
\mathsf{cst}(a,b,1) &= 3t_{\text{app}} + t_{\leq} + t_{\text{rec}} + t_{\text{true}} \\
\mathsf{cst}(a,b,n) &= \mathsf{cst}(b, a+b, n-1) + 3t_{\text{app}} + t_{\leq} + t_{\text{rec}} + t_{\text{false}} + t_{-} + t_{+} \quad (n > 1) \\
\mathsf{val}(a,b,1) &= a \\
\mathsf{val}(a,b,n) &= \mathsf{val}(b, a+b, n-1) \quad (n-1)
\end{aligned}$$

from which it follows by an easy inductive argument that

$$\begin{aligned}
\mathsf{cst}(a,b,n) &= n(3t_{\text{app}} + t_{\leq} + t_{\text{rec}}) + (n-1)(t_{-} + t_{\text{false}} + t_{+}) + t_{\text{true}} \\
\mathsf{val}(a,b,1) &= a \\
\mathsf{val}(a,b,2) &= b \\
\mathsf{val}(a,b,n) &= \mathsf{fb}_v(n-2)a + \mathsf{fb}_v(n-1)a \quad (n > 2) \\
\mathsf{val}(1,1,n) &= \mathsf{fb}_v(n)
\end{aligned}$$

Therefore `fib2` does return the Fibonacci function, `ifib` and `fib2` are linear in n, and $\mathsf{cst}(a,b,n)$ is independent of a and b.

3.8.6 The `twice` program

Our next example, which includes higher-order types, is the program

$$\texttt{twice} = \texttt{lam f.lam x.f(f(x))}$$

Let $y_f : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau \to \tau]\!]$ represent the value assigned to `f` and let $y_x : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ represent the value assigned to `x`. Then

$$\begin{aligned}\mathbf{apply}(\mathcal{V}[\![\texttt{twice}]\!], y_f, y_x) &= [\![2t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\texttt{f} : \tau \to \tau, \texttt{x} : \tau \vdash \texttt{f(f(x))} : \tau]\!] \circ \langle\rangle(\eta \circ y_f, \eta \circ y_x) \\ &= [\![2t_{\mathrm{app}}]\!] \circ \mathbf{capply}(\eta \circ y_f, \mathbf{apply}(\eta \circ y_f, y_x))\end{aligned}$$

To simplify this further, we need to acquire further information about the behavior of y_f. Certainly, if $\mathbf{apply}(\eta \circ y_f, y_x)$ is $\perp$, the meaning of the entire expression is $\perp$. Also, if y_f represents a constant function, i.e., if for all $y : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$, $\mathbf{apply}(\eta \circ y_f, y) = [\![t]\!] \circ \eta \circ k$ for some $k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$, then

$$\begin{aligned}\mathbf{apply}(\mathcal{V}[\![\texttt{twice}]\!], y_f, y_x) &= [\![2t_{\mathrm{app}}]\!] \circ \mathbf{capply}(\eta \circ y_f, [\![t]\!] \circ \eta \circ k) \\ &= [\![2t + 2t_{\mathrm{app}}]\!] \circ \eta \circ k\end{aligned}$$

Thus for constant input functions the cost of applying `twice` is the same as applying the function twice (regardless of the input).

For a more general solution, suppose that, for any morphism $y : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$,

$$\mathbf{apply}(\eta \circ y_f, y) = [\![\mathsf{cst}(y)]\!] \circ \eta \circ \mathsf{val}(y)$$

where cst is a function from morphisms to costs representing the cost of applying y_f, and val is a function from morphisms to morphisms that returns the final value (without costs) of applying y_f.

$$\mathbf{apply}(\mathcal{V}[\![\texttt{twice}]\!], y_f, y_x) = [\![2t_{\mathrm{app}} + \mathsf{cst}(y_x) + \mathsf{cst}(\mathsf{val}(y_x))]\!] \circ \eta \circ \mathsf{val}(\mathsf{val}(y_x))$$

This version succinctly shows how `twice` generates costs. As expected, the cost of `twice` is the cost of calculating a function f on the original input y and on the input $f(y)$ plus two applications.

To see how this works with a specific example, let us revisit the Fibonacci function `fib2` from the previous section. With that function, we have that for any $\mathsf{n}_n{}^I : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$,

$$\begin{aligned}\mathsf{cst}(\mathsf{n}_n{}^I) &= nc_1 + (n-1)c_2 + t_{\mathrm{true}} \\ \mathsf{val}(\mathsf{n}_n{}^I) &= \mathsf{fb}_v(n)\end{aligned}$$

Where $c_1 = 3t_{\mathrm{app}} + t_{\leq} + t_{\mathrm{rec}}$, $c_2 = t_{-} + t_{\mathrm{false}} + t_{+}$, and $\mathsf{fb}_v(n)$ is the Fibonacci function. From the above we thus know that

$$\begin{aligned}\mathbf{apply}(\mathcal{V}[\![\texttt{twice}]\!], y_{\mathrm{fib}}, \mathsf{n}_n{}^I) &= [\![2t_{\mathrm{app}} + nc_1 + (n-1)c_2 + t_{\mathrm{true}} + \mathsf{fb}_v(n)c_1 + (\mathsf{fb}_v(n) - 1)c_2 + t_{\mathrm{true}}]\!] \\ &\qquad \circ \eta \circ \mathsf{n}_{\mathrm{fb}_v(\mathrm{fb}_v(n))}{}^I \\ &= [\![2t_{\mathrm{app}} + (n + \mathsf{fb}_v(n))c_1 + (n - 1 + \mathsf{fb}_v(n-1))c_2 + 2t_{\mathrm{true}}]\!] \\ &\qquad \circ \eta \circ \mathsf{n}_{\mathrm{fb}_v(\mathrm{fb}_v(n))}{}^I\end{aligned}$$

Thus the cost of `twice(fib2)` is proportional to the Fibonacci value of the argument.

3.8.7 Creating lists with `tabulate`

Our final example contains higher-order functions and recursion. Let

$$\begin{aligned}\texttt{itab} &= \texttt{rec t.lam i.lam f.lam n.if i} \leq n \texttt{ then (f(i))::(t (i} + \bar{1}\texttt{) f n) else nil} \\ \texttt{tabulate} &= \texttt{itab}(\bar{1})\end{aligned}$$

The program `tabulate` takes a function `f` and an integer n and creates a list $[\texttt{f}(1), \ldots, \texttt{f}(n)]$. We will need more information about the behavior of `f` to fully calculate the cost of `tabulate`, but some parts of the calculation are independent of the actual function.

First, if $i > n$, then

$$\mathbf{apply}(\mathtt{itab}, \mathsf{n}_i{}^I, z_f, \mathsf{n}_n{}^I) = [\![t_{\mathrm{false}} + t_{\leq} + 3t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{nil}^I$$

Otherwise,

$$\begin{aligned}\mathbf{apply}(\mathtt{itab}, \mathsf{n}_i{}^I, z_f, \mathsf{n}_n{}^I) \quad = \quad & [\![t_{\mathrm{true}} + t_{\leq} + 3t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathbf{capply}(\mathcal{V}[\![\mathsf{cons}]\!], \mathbf{apply}(\eta \circ z_f, \mathsf{n}_i{}^I), \\ & [\![t_+]\!] \circ \mathbf{apply}(\mathtt{itab}, \mathsf{n}_{i+1}{}^I, z_f, \mathsf{n}_n{}^I))\end{aligned}$$

By induction on $n - j$, we can show that if, for some $i \leq j \leq n$, $\mathbf{apply}(\eta \circ z_f, \mathsf{n}_j{}^I) = \perp$, then $\mathbf{apply}(\mathtt{itab}, \mathsf{n}_i{}^I, f, \mathsf{n}_n{}^I) = \perp$ as well. Therefore, for any interesting examples, we should assume that $\mathbf{apply}(\eta \circ f, \mathsf{n}_j{}^I) \neq \perp$ for any j, $i \leq j \leq n$. From that assumption we know that there exist functions cst and val such that

$$\mathbf{apply}(\eta \circ f, \mathsf{n}_j{}^I) = [\![\mathsf{cst}(j)]\!] \circ \eta \circ \mathsf{val}(j)$$

Thus, when $i \leq n$,

$$\begin{aligned}\mathbf{apply}(\mathtt{itab}, \mathsf{n}_i{}^I, f, \mathsf{n}_n{}^I) \quad = \quad & [\![\mathsf{cst}(i) + t_{\mathrm{true}} + t_{\leq} + 3t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathbf{capply}(\mathcal{V}[\![\mathsf{cons}]\!], \eta \circ \mathsf{val}(i), \\ & [\![t_+]\!] \circ \mathbf{apply}(\mathtt{itab}, \mathsf{n}_{i+1}{}^I, f, \mathsf{n}_n{}^I))\end{aligned}$$

If we further assume that `itab` terminates, then there exist functions $\mathsf{tb}_t(i, n)$ and $\mathsf{tb}_v(i, n)$ such that

$$\mathbf{apply}(\mathtt{itab}, \mathsf{n}_i{}^I, f, \mathsf{n}_n{}^I) = [\![\mathsf{tb}_t(i, n)]\!] \circ \eta \circ \mathsf{tb}_v(i, n)$$

Then

$$\begin{aligned}&\mathbf{apply}(\mathtt{itab}, \mathsf{n}_i{}^I, f, \mathsf{n}_n{}^I) \\ &\quad = [\![\mathsf{tb}_t(i+1, n) + \mathsf{cst}(i) + t_+ + t_{\mathrm{true}} + t_{\leq} + 3t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{cons}^I \\ &\qquad\qquad \circ \langle \mathsf{val}(i), \mathsf{tb}_v(i+1, n)\rangle\end{aligned}$$

Thus we get the equations

$$\begin{aligned}\mathsf{tb}_t(i, n) = \quad & t_{\mathrm{false}} + t_{\leq} + 3t_{\mathrm{app}} + t_{\mathrm{rec}} && i > n \\ \mathsf{tb}_t(i, n) = \quad & \mathsf{tb}_t(i+1, n) + \mathsf{cst}(i) + t_+ + t_{\mathrm{true}} + t_{\leq} + 3t_{\mathrm{app}} + t_{\mathrm{rec}} && i \leq n \\ \mathsf{tb}_v(i, n) = \quad & \mathsf{nil}^I && i > n \\ \mathsf{tb}_v(i, n) = \quad & \mathsf{cons}^I \circ \langle \mathsf{val}(i), \mathsf{tb}_v(i+1, n)\rangle && i \leq n\end{aligned}$$

Solving these equations leads to the general forms (for $i \leq n + 1$):

$$\begin{aligned}\mathsf{tb}_t(i, n) = \quad & t_{\mathrm{false}} + (n - i + 2)(t_{\leq} + 3t_{\mathrm{app}} + t_{\mathrm{rec}}) + (n - i + 1)(t_+ + t_{\mathrm{true}}) + \textstyle\sum_{j=i}^{n} \mathsf{cst}(j) \\ \mathsf{tb}_v(i, n) = \quad & [\mathsf{val}(i), \ldots, \mathsf{val}(n)]\end{aligned}$$

For `tabulate`, $i = 1$, therefore

$$\begin{aligned}\mathsf{tb}_t(1, n) = \quad & t_{\mathrm{false}} + (n + 1)(t_{\leq} + 3t_{\mathrm{app}} + t_{\mathrm{rec}}) + n(t_+ + t_{\mathrm{true}}) + \textstyle\sum_{j=1}^{n} \mathsf{cst}(j) \\ \mathsf{tb}_v(1, n) = \quad & [\mathsf{val}(1), \ldots, \mathsf{val}(n)]\end{aligned}$$

which means that

$$\begin{aligned}
&\mathbf{apply}(\mathcal{V}[\![\texttt{tabulate}]\!], f, \mathsf{n}_n{}^I) \\
&\qquad = \mathbf{apply}(\mathcal{V}[\![\texttt{itab}, \mathsf{n}_1{}^I, f, \mathsf{n}_n{}^I]\!]) \\
&\qquad = [\![t_{\text{false}} + (n+1)(t_{\leq} + 3t_{\text{app}} + t_{\text{rec}}) + n(t_+ + t_{\text{true}}) + \textstyle\sum_{j=1}^{n} \mathsf{cst}(j)]\!] \circ \eta \circ [\mathsf{val}(1), \ldots, \mathsf{val}(n)]
\end{aligned}$$

From this we can see that the the cost of `tabulate` is linear except for the costs of applying f to the arguments of the list. In particular, if we apply `tabulate` to `fib2` from section 3.8.5, then $\eta \circ f = \mathcal{V}[\![\texttt{fib2}]\!]$, so $\mathsf{cst}(i) = i(3t_{\text{app}} + t_{\leq} + t_{\text{rec}}) + (i-1)(t_- + t_{\text{false}} + t_+) + t_{\text{true}}$. Therefore,

$$\sum_{j=1}^{n} \mathsf{cst}(j) = \frac{n(n+1)}{2}(3t_{\text{app}} + t_{\leq} + t_{\text{rec}}) + \frac{n(n-1)}{2}(t_- + t_{\text{false}} + t_+) + nt_{\text{true}}$$

which means that

$$\mathbf{capply}(\mathcal{V}[\![\texttt{tabulate}]\!], \mathcal{V}[\![\texttt{fib2}]\!], \eta \circ \mathsf{n}_n{}^I) = [\![t]\!] \circ \eta \circ [\mathrm{fib}(1), \ldots, \mathrm{fib}(n)]$$

where

$$t = 2n(t_{\text{true}}) + \frac{(n^2 - n + 2)}{2} t_{\text{false}} + \frac{n(n+1)}{2} t_+ + \frac{n(n-1)}{2} t_- + \frac{(n+1)(n+2)}{2}(t_{\leq} + 3t_{\text{app}} + t_{\text{rec}})$$

i.e., `tabulate fib2` is quadratic overall.

3.9 Relating programs

In addition to analyzing individual programs, we may want to compare two different programs or expressions. For example, we may want to know if program A is always faster than program B, sometime faster, or always slower. We may also want to know if a certain program transformation improves the overall speed, or, if the improvement is only occasional, under what conditions improvement occurs.

In order to compare two programs, however, we need to compare costs. We could simply state that $t_1 \leq t_2$ if for some cost t, $t_2 = t_1 + t$; other orderings, however, may also be useful for specific purposes (such as one that weighs different base costs separately such that, for example, t_{app} becomes less than t_{true}). Therefore let $\preceq$ be a *cost ordering* if it is a preorder and if $t_1 \preceq t_2$ implies that, for all costs t, $t_1 + t \preceq t_2 + t$ and $t + t_1 \preceq t + t_2$. This assumes seems very natural and is likely to hold for any reasonable ordering. We want an ordering to be a preorder because a non-transitive ordering is not likely to be sensible; it is also easier to state and prove properties if we can assume that $t \preceq t$. In practice $\preceq$ will be a partial order when $+$ is commutative; it is rarely necessary for different costs to be equivalent except when the order of the costs is significant.

If there were no internal costs then we could simply state that given morphisms $z_1, z_2 : \mathbf{1} \to CA$, $z_1 \preceq z_2$, if either $z_1 = z_2 = \perp$, or if $z_1 = [\![t_1]\!] \circ \eta \circ y$ and $z_2 = [\![t_2]\!] \circ \eta \circ y$ where $t_1 \preceq t_2$. This definition is insufficient for any expression containing internal cost. Therefore we will define an *improvement relation* $\preceq_\tau^{\mathrm{V}}$ recursively by type for morphisms from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ (and extend it to related morphisms).

To avoid problems caused by inexpressible functions in the mathematical space (such as one that gives different results based on internal cost), we will also impose a monotonicity requirement. The definition of "monotone" depends on both the particular improvement relation $\preceq_\tau^{\mathrm{V}}$ and τ, leading to a pair of mutually recursive definitions. We show later in this section that the meanings of all well-typed expressions are monotone.

Although the definition of $\preceq^{\mathrm{V}}_{\tau}$ has several possible uses, this section contains only some very basic ones. Primarily, we would like $\preceq^{\mathrm{V}}_{\tau}$ to be closely related to its operational equivalent. Operationally, for ground types and closed expressions, let $e_1 \trianglelefteq^{\mathrm{V}} e_2$ mean that $e_1 \stackrel{t_1}{\Rightarrow}_{\mathrm{v}} v$ if and only if $e_1 \stackrel{t_1}{\Rightarrow}_{\mathrm{v}} v$ with $t_1 \preceq t_2$. Note that requires that the final value v be the same; we are only interested in comparing extensionally equivalent expression. If ground type values are defined reasonably, in that $\mathcal{V}[\![v]\!] = \mathcal{V}[\![v']\!]$ implies that $v = v'$, then this definition is equivalent to the definition of $\preceq^{\mathrm{V}}_{\tau}$ for ground types. For higher order types and expressions with free variables, we use contexts. A context $C[\,]$ is an expression with a "hole", such as $\mathtt{lam}\ x.[\,] + \bar{1}$. The expression $C[e_1]$ fills the hole with the expression e_1, i.e., $(\mathtt{lam}\ x.[\,] + \bar{1})[e_1] = \mathtt{lam}\ x.e_1 + \bar{1}$. Filling the hole is similar to substitution, but does not avoid trapping bound variables. Thus $[x/y](\mathtt{lam}\ x.y) = \mathtt{lam}\ z.x$, but $(\mathtt{lam}\ x.[\,])[x] = \mathtt{lam}\ x.x$. With contexts we can say that $e_1 \trianglelefteq^{\mathrm{V}} e_2$ if for all contexts $C[\,]$ such that $C[e_1]$ and $C[e_2]$ are closed expressions of ground type, $C[e_1] \trianglelefteq^{\mathrm{V}} C[e_2]$. It is generally difficult to prove that $\trianglelefteq^{\mathrm{V}}$ holds for higher order expressions, but for most cases we can prove it via the denotational definition for we can show that $\mathcal{V}[\![e_1 : \tau]\!] \preceq^{\mathrm{V}}_{\tau} \mathcal{V}[\![e_2 : \tau]\!]$ implies that $e_1 \trianglelefteq^{\mathrm{V}} e_2$.

A few functions on morphisms are useful for defining monotonicity and the relation $\preceq^{\mathrm{V}}_{\tau}$. For any morphism $z : \mathbf{1} \to CA$, if, for some cost t, $z = [\![t]\!] \circ \eta \circ y$, then let $\mathsf{val}(z) = y$ and $\mathsf{cost}(z) = t$. Thus val and cost are partial functions that give us the external cost and intensional value, when such exist. As we can safely assume, because of adequacy, throughout this section that a morphism $z : \mathbf{1} \to CA$ is either $\perp$ or is of the form $[\![t]\!] \circ \eta \circ y$, val and cost are undefined only on $\perp$.

Lastly, we need to be able to examine the parts of a constructed data type. To prove adequacy for the extensional semantics, we examined the parts of a constructed data type through the use of projections ρ_{i_j}. Along with these projections was a method for comparing the overall structure of a data type, namely composition with $F_\delta(!, \ldots, !)$. Therefore assume that, given a type $\tau = \delta(\tau_1, \ldots, \tau_n)$, for each extensional projection $\rho_{i_j} : F_\delta(\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!], \ldots, \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_n]\!]) \to L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_i]\!]$ there exists an intensional projection $\rho^{\mathrm{I}}_{i_j} : F^{\mathrm{V}}_\delta(\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!] \to \mathcal{T}^{\mathrm{V}}[\![\tau_n]\!]) \to C\mathcal{T}^{\mathrm{V}}[\![\tau_i]\!]$ such that $E\rho^{\mathrm{I}}_{i_j} = \rho_{i_j}$. For a cost structure derived from an arrow cost structure, $\rho^{\mathrm{I}}_{i_j} = (L\eta^A \circ \rho_{i_j}, \rho_{i_j})$; this is a valid morphism because ρ_{i_j} is a natural transformation. For the structure comparison, we can use $F^{\mathrm{V}}_\delta(!, \ldots, !)$, the intensional version of $F_\delta(!, \ldots, !)$.

We define monotonicity and $\preceq^{\mathrm{V}}_{\tau}$ by mutual induction on the structure of τ:

Definition 3.9.1 For each type τ and a cost order $\preceq$, a morphism $z : \mathbf{1} \to C\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ is *monotone* relative to $\preceq$ if either $z = \perp$, or, for some cost t, $z = [\![t]\!] \circ \eta \circ y$, where y is monotone relative to $\preceq$ if one of the following holds:

- τ is a ground type
- $\tau = \delta(\tau_1, \ldots, \tau_n)$, and for all intensional projections $\rho^{\mathrm{I}}_{i_j}$, $\rho^{\mathrm{I}}_{i_j} \circ y$ is monotone relative to $\preceq$.
- $\tau = \tau_1 \to \tau_2$ and for all monotone $y_1, y_2 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau_1]\!]$, $\mathbf{apply}(\eta \circ y, y_1)$ and $\mathbf{apply}(\eta \circ y, y_2)$ are monotone relative to $\preceq$, and $y_1 \preceq^{\mathrm{V}}_{\tau_1} y_2$ implies that

$$\mathbf{apply}(\eta \circ y, y_1) \preceq^{\mathrm{V}}_{\tau_2} \mathbf{apply}(\eta \circ y, y_2)$$

When the cost order $\preceq$ is understood, we simply say that z is monotone.

Definition 3.9.2 For any type environment Γ, a morphism $\langle\rangle(z_1, \ldots, z_n) : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]$ is monotone if each z_i is monotone. A morphism $g : \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!] \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ is monotone if for all monotone $r, r' : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]$, $g \circ r$ is monotone and if whenever $r \preceq^{\mathrm{V}}_{\Gamma} r'$, $g \circ r \preceq^{\mathrm{V}}_{\tau} g \circ r'$.

Similarly, a morphism $\langle\rangle(y_1,\ldots,y_n) : \mathbf{1} \to \times(\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!],\ldots,\mathcal{T}^{\mathrm{V}}[\![\tau_n]\!])$ is monotone if each y_i is monotone, and a morphism $g : \times(\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!],\ldots,\mathcal{T}^{\mathrm{V}}[\![\tau_n]\!]) \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ is monotone if for all monotone $x, x' : \mathbf{1} \to \times(\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!],\ldots,\mathcal{T}^{\mathrm{V}}[\![\tau_n]\!])$, $g \circ x$ is monotone and if whenever $x \preceq^{\mathrm{V}}_{\tau_1,\ldots,\tau_n} x'$, $g \circ x \preceq^{\mathrm{V}}_{\tau} g \circ x'$.

With the definition of monotonicity, we now define the improvement relation:

Definition 3.9.3 Given two morphisms $y_1, y_2 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ that are monotone relative to $\preceq$, $y_1 \preceq^{\mathrm{V}}_{\tau} y_2$ if

- τ is a ground type and $y_1 = y_2$.
- $\tau = \delta(\tau_1,\ldots,\tau_n)$, $F^{\mathrm{V}}_{\delta}(!,\ldots,!) \circ y_1 = F^{\mathrm{V}}_{\delta}(!,\ldots,!) \circ y_2$, and, for each projection $\rho^{\mathrm{I}}_{i_j}$

$$\rho^{\mathrm{I}}_{i_j} \circ y_1 \preceq^{\mathrm{V}}_{\tau_i} \rho^{\mathrm{I}}_{i_j} \circ y_2$$

- $\tau = \tau_1 \to \tau_2$, and, for all monotone morphisms $y : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau_1]\!]$,

$$\mathbf{apply}(\eta \circ y_1, y) \preceq^{\mathrm{V}}_{\tau_2} \mathbf{apply}(\eta \circ y_2, y)$$

Definition 3.9.4 For any monotone morphisms $z_1, z_2 : \mathbf{1} \to C\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$, $z_1 \preceq^{\mathrm{V}}_{\tau_i} z_2$ if $\mathsf{val}(z_1)$ exists if and only if $\mathsf{val}(z_2)$ exists and, when they exist, $\mathsf{cost}(z_1) \preceq \mathsf{cost}(z_2)$, and $\mathsf{val}(z_1) \preceq^{\mathrm{V}}_{\tau} \mathsf{val}(z_2)$.

For any type environment Γ and morphisms $\langle\rangle(z_1,\ldots,z_n), \langle\rangle(z'_1,\ldots,z'_n) : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]$, we say that $\langle\rangle(z_1,\ldots,z_n) \preceq^{\mathrm{V}}_{\Gamma} \langle\rangle(z'_1,\ldots,z'_n)$ if, for each $1 \le i \le n$, $z_i \preceq^{\mathrm{V}}_{\tau_i} z'_i$. For $g_1, g_2 : \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!] \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$, $g_1 \preceq^{\mathrm{V}}_{\tau} g_2$ if, for all monotone $r : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]$, $g_1 \circ r \preceq^{\mathrm{V}}_{\tau} g_2 \circ r$.

Similarly, for any morphisms $\langle\rangle(y_1,\ldots,y_n), \langle\rangle(y'_1,\ldots,y'_n) : \mathbf{1} \to \times(\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!],\ldots,\mathcal{T}^{\mathrm{V}}[\![\tau_n]\!])$, we say that $\langle\rangle(y_1,\ldots,y_n) \preceq^{\mathrm{V}}_{\tau_1,\ldots,\tau_n} \langle\rangle(y'_1,\ldots,y'_n)$ if, for each $1 \le i \le n$, $y_i \preceq^{\mathrm{V}}_{\tau_i} y'_i$. Given two morphisms $g_1, g_2 : \times(\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!],\ldots,\mathcal{T}^{\mathrm{V}}[\![\tau_n]\!]) \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$, $g_1 \preceq^{\mathrm{V}}_{\tau} g_2$ if, for all monotone $x : \mathbf{1} \to \times(\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!],\ldots,\mathcal{T}^{\mathrm{V}}[\![\tau_n]\!])$, $g_1 \circ x \preceq^{\mathrm{V}}_{\tau} g_2 \circ x$.

It can be easily shown, by induction on the structure of τ, that $\preceq^{\mathrm{V}}_{\tau}$ is a preorder for each type τ.

As with soundness and adequacy, we will need to make some assumptions about the behavior of constants. Because of the way we defined monotonicity, the only assumption needed is that $\mathcal{C}^{\mathrm{V}}[\![c : \tau]\!]$ is monotone for each constant c of type τ. It is straightforward to show that raise preserves monotonicity.

It follows directly from the definition that if $z : \mathbf{1} \to C\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ is monotone, so is $[\![t]\!] \circ z$. Similarly, by our definition of cost orderings, if $t \preceq t'$, then $[\![t]\!] \circ z \preceq^{\mathrm{V}}_{\tau} [\![t']\!] \circ z$, and if $z \preceq^{\mathrm{V}}_{\tau} z'$, then $[\![t]\!] \circ z \preceq^{\mathrm{V}}_{\tau} [\![t]\!] \circ z'$.

It is slightly more difficult to prove that fixed points preserve monotonicity. We can show this given the following lemma:

Lemma 3.9.1 *Let the sequence $y_1 \le y_2 \le y_3 \le \ldots$ be monotone morphisms from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ and let the sequence $z_1 \le z_2 \le z_3 \le \ldots$ be monotone morphisms from $\mathbf{1}$ to $C\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$. Then the following four properties hold:*

1. *$\bigsqcup_{n=1}^{\infty} y_n$ is monotone.*
2. *$\bigsqcup_{n=1}^{\infty} z_n$ is monotone*

3. *For all monotone morphisms $y'_1 \leq y'_2 \leq \ldots$ from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ such that for each n, $y_n \preceq^{\mathrm{V}}_{\tau} y'_n$, $\bigsqcup_{n=1}^{\infty} y_n \preceq^{\mathrm{V}}_{\tau} \bigsqcup_{n=1}^{\infty} y'_n$.*
4. *For all monotone morphisms $z'_1 \leq z'_2 \leq \ldots$ from $\mathbf{1}$ to $C\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ such that for each n, $z_n \preceq^{\mathrm{V}}_{\tau} z'_n$, $\bigsqcup_{n=1}^{\infty} z_n \preceq^{\mathrm{V}}_{\tau} \bigsqcup_{n=1}^{\infty} z'_n$.*

Proof. For each type τ we can show that #1 implies #2 and #3 implies #4. To show that #1 implies #2, from the ordering requirement for cost structures, we know that either $z_n = \perp$ for all n or there exists a cost t and an integer N such that for all $n \geq N$, there exists a monotone $x_n : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ with $z_n = [\![t]\!] \circ \eta \circ x_n$. Thus

$$\bigsqcup_{n=1}^{\infty} z_n = \bigsqcup_{n=N}^{\infty} ([\![t]\!] \circ \eta \circ x_n) = [\![t]\!] \circ \eta \circ \bigsqcup_{n=N}^{\infty} x_n$$

Parts #1 and #3 are proven by induction on τ. They may require parts #2 and #4, but only on smaller types. □

With this lemma, we can now prove that all expressions are monotone.

Theorem 3.9.2 *Let $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ be a type environment and suppose that $\Gamma \vdash e : \tau$. Then $\mathcal{V}[\![\Gamma \vdash e : \tau]\!]$ is monotone.*

Proof. Let $r, r' : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]$ be monotone with $r \preceq^{\mathrm{V}}_{\Gamma} r'$. Then $\mathcal{V}[\![\Gamma \vdash e : \tau]\!]$ is monotone if $\mathcal{V}[\![\Gamma \vdash e : \tau]\!] \circ r$ is monotone and if

$$\mathcal{V}[\![\Gamma \vdash e : \tau]\!] \circ r \preceq^{\mathrm{V}}_{\tau} \mathcal{V}[\![\Gamma \vdash e : \tau]\!] \circ r'$$

We show this by induction on the structure of e. All cases are straightforward; we list a few of the more complex ones here.

Case $e = \mathtt{lam}\ x.e'$**,** $\tau = \tau_1 \to \tau_2$**:**

Let $f_0 = [\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau_1 \vdash e' : \tau_2]\!] \circ (\mathsf{id} \times \eta)$. Then $\mathcal{V}[\![\Gamma \vdash e : \tau]\!] = \eta \circ \mathsf{curry}(f_0)$. Therefore to show that $\mathcal{V}[\![\Gamma \vdash e : \tau]\!]$ is monotone, we need only to show that $\mathsf{curry}(f_0) \circ r : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau_1 \to \tau_2]\!]$ is monotone, and that $\mathsf{curry}(f_0) \circ r \preceq^{\mathrm{V}}_{\tau_1 \to \tau_2} \mathsf{curry}(f_0) \circ r'$. Let y_1 and y_2 be monotone morphisms from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\tau_1]\!]$ such that $y_1 \preceq^{\mathrm{V}}_{\tau_1} y_2$. First, note that

$$\begin{aligned} \mathbf{apply}(\eta \circ \mathsf{curry}(f_0) \circ r, y_1) &= \mathsf{app} \circ \langle \mathsf{curry}(f_0) \circ r, y_1 \rangle \\ &= [\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau_1 \vdash e' : \tau_2]\!] \circ \langle r, \eta \circ y_1 \rangle \end{aligned}$$

Similarly,

$$\mathbf{apply}(\eta \circ \mathsf{curry}(f_0) \circ r, y_2) = [\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau_1 \vdash e' : \tau_2]\!] \circ \langle r, \eta \circ y_2 \rangle$$

It is clear that $\langle r, \eta \circ y_1 \rangle$ and $\langle r, \eta \circ y_2 \rangle$ are monotone. Therefore, $\mathbf{apply}(\eta \circ \mathsf{curry}(f_0) \circ r, y_1)$ and $\mathbf{apply}(\eta \circ \mathsf{curry}(f_0) \circ r, y_2)$ are monotone with

$$\mathbf{apply}(\eta \circ \mathsf{curry}(f_0) \circ r, y_1) \preceq^{\mathrm{V}}_{\tau_2} \mathbf{apply}(\eta \circ \mathsf{curry}(f_0) \circ r, y_2)$$

Thus $\mathcal{V}[\![\Gamma \vdash e : \tau]\!] \circ r$ is monotone. For the second part, let $y : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\tau_1]\!]$ be monotone. Then

$$\begin{aligned} \mathbf{apply}(\eta \circ \mathsf{curry}(f_0) \circ r, y) &= [\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau_1 \vdash e' : \tau_2]\!] \circ \langle r, \eta \circ y \rangle \\ &\preceq^{\mathrm{V}}_{\tau_2} [\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau_1 \vdash e' : \tau_2]\!] \circ \langle r', \eta \circ y \rangle \\ &= \mathbf{apply}(\eta \circ \mathsf{curry}(f_0) \circ r', y) \end{aligned}$$

Thus $\eta \circ \mathsf{curry}(f_0) \circ r \preceq^{\mathrm{V}}_{\tau_1 \to \tau_2} \eta \circ \mathsf{curry}(f_0) \circ r'$, i.e., $\mathcal{V}[\![\Gamma \vdash e : \tau]\!] \circ r \preceq^{\mathrm{V}}_{\tau} \mathcal{V}[\![\Gamma \vdash e : \tau]\!] \circ r'$. Therefore $\mathcal{V}[\![\Gamma \vdash e : \tau]\!]$ is monotone.

Case $e = \texttt{rec}\ x.e'$**:**

$\mathcal{V}[\![\Gamma \vdash e : \tau]\!] = \mathsf{fixp}([\![t_{\text{rec}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau \vdash e' : \tau]\!])$. By the induction hypothesis $\mathcal{V}[\![\Gamma, x : \tau \vdash e' : \tau]\!]$ is monotone, therefore so is $[\![t_{\text{rec}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau \vdash e' : \tau]\!]$. We therefore only need to show that applying the fixed point operator preserves monotonicity. Let $f = [\![t_{\text{rec}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau \vdash e' : \tau]\!]$

By the definition of fixp, $\mathsf{fixp}(f)$ is the limit of the following morphisms:

$$\begin{aligned} f_0 = &\quad \bot_{\mathcal{T}^{\mathrm{V}}[\![\tau]\!]} \circ !_{\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]} \\ f_n = &\quad f \circ \langle \mathsf{id}_{\mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]}, f_{n-1} \rangle \end{aligned}$$

We know immediately that $f_0 = f_0 \circ r$ is monotone, and, by the induction hypothesis, if $f_{n-1} \circ r$ is monotone, then so is $f_n \circ r$. Therefore by the first half of Lemma 3.9.1,

$$(\bigsqcup_{n=0}^{\infty} f_n) \circ r = \mathsf{fixp}(f) \circ r$$

is monotone as well. Therefore $\mathcal{V}[\![\Gamma \vdash e : \tau]\!] \circ r$ is monotone.

Furthermore, $f_0 \circ r = f_0 \circ r' = \bot$ so $f_0 \circ r \preceq_\tau^{\mathrm{V}} f_0 \circ r'$. If $f_n \circ r \preceq_\tau^{\mathrm{V}} f_n \circ r'$, then $f_{n+1} \circ r \preceq_\tau^{\mathrm{V}} f_{n+1} \circ r'$ as well. Thus by the second half of Lemma 3.9.1,

$$\mathsf{fixp}(f) \circ r = \bigsqcup_{n=0}^{\infty} f_n \circ r \preceq_\tau^{\mathrm{V}} \bigsqcup_{n=0}^{\infty} f_n \circ r' = \mathsf{fixp}(f) \circ r'$$

Therefore $\mathcal{V}[\![\Gamma \vdash e : \tau]\!] \circ r \preceq_\tau^{\mathrm{V}} \mathcal{V}[\![\Gamma \vdash e : \tau]\!] \circ r'$, so $\mathcal{V}[\![\Gamma \vdash e : \tau]\!]$ is monotone.

□

Now that we have a definition of $\preceq_\tau^{\mathrm{V}}$ that is known to apply to expressions, we can show that $\preceq_\tau^{\mathrm{V}}$ is also sound relative to the operational version; namely that, for any closed expressions e_1, e_2 of type τ, $\mathcal{V}[\![e_1 : \tau]\!] \preceq_\tau^{\mathrm{V}} \mathcal{V}[\![e_2 : \tau]\!]$ implies that $e_1 \trianglelefteq^{\mathrm{V}} e_2$. This property follows directly from another property that is useful in its own right: the relation $\preceq_\tau^{\mathrm{V}}$ is compositional, which means that we can determine that two expression relate by $\preceq_\tau^{\mathrm{V}}$ by showing that their subparts relate.

Theorem 3.9.3 *Suppose that for closed expression e_1 and e_2 of type τ, $\mathcal{V}[\![e_1 : \tau]\!] \preceq_\tau^{\mathrm{V}} \mathcal{V}[\![e_2 : \tau]\!]$. Then for all contexts $C[\,]$, if, for some type environment Γ, $\Gamma \vdash C[e_1] : \tau'$, then*

$$\mathcal{V}[\![\Gamma \vdash C[e_1] : \tau']\!] \preceq_{\tau'}^{\mathrm{V}} \mathcal{V}[\![\Gamma \vdash C[e_2] : \tau']\!]$$

Proof. By induction on the structure of $C[\,]$. First, note that if $\Gamma \vdash C[e_1] : \tau'$, then $\Gamma \vdash C[e_2] : \tau'$. Next, while in general $C[\,]$ is assumed to have a hole, its subexpressions may not. For expressions without a hole, however, $C[e_1] = C[e_2]$, so the theorem holds trivially. Similarly the theorem holds immediately when $C[\,] = [\,]$. Therefore we can assume that the theorem holds on subexpressions without having to worry about which subexpression has the hole.

For the other cases, let $r : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!]$ be monotone. Then

$$\mathcal{V}[\![\Gamma \vdash C[e_1] : \tau']\!] \preceq_{\tau'}^{\mathrm{V}} \mathcal{V}[\![\Gamma \vdash C[e_2] : \tau']\!]$$

when

$$\mathcal{V}[\![\Gamma \vdash C[e_1] : \tau']\!] \circ r \preceq_{\tau'}^{\mathrm{V}} \mathcal{V}[\![\Gamma \vdash C[e_2] : \tau']\!] \circ r$$

All cases are straightforward; we show a few here.

$C[\,] = \texttt{if } C_1[\,] \texttt{ then } C_2[\,] \texttt{ else } C_3[\,]$

By the induction hypothesis $\mathcal{V}[\![\Gamma \vdash C_1[e_1] : \mathbf{bool}]\!] \circ r \preceq^{\mathrm{V}}_{\mathbf{bool}} \mathcal{V}[\![\Gamma \vdash C_2[e_2] : \mathbf{bool}]\!] \circ r$. If $\mathcal{V}[\![\Gamma \vdash C_1[e_1] : \mathbf{bool}]\!] \circ r = \bot$, then $\mathcal{V}[\![\Gamma \vdash C_2[e_2] : \mathbf{bool}]\!] \circ r = \bot$ as well, and

$$\begin{array}{cl} \multicolumn{2}{l}{\mathcal{V}[\![\Gamma \vdash \texttt{if } C_1[e_1] \texttt{ then } C_2[e_1] \texttt{ else } C_3[e_1] : \tau']\!] \circ r} \\ = & \bot \\ \preceq^{\mathrm{V}}_{\tau'} & \bot = \mathcal{V}[\![\Gamma \vdash \texttt{if } C_1[e_2] \texttt{ then } C_2[e_2] \texttt{ else } C_3[e_2] : \tau']\!] \circ r \end{array}$$

Otherwise there exists costs t and t', and morphism $b : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{bool}]\!]$, where b is either tt or ff, such that $t \preceq t'$, $\mathcal{V}[\![\Gamma \vdash C_1[e_1] : \mathbf{bool}]\!] \circ r = [\![t]\!] \circ \eta \circ b$, and $\mathcal{V}[\![\Gamma \vdash C_1[e_2] : \mathbf{bool}]\!] \circ r = [\![t']\!] \circ \eta \circ b$

If b is tt, then

$$\begin{array}{cl} \multicolumn{2}{l}{\mathcal{V}[\![\Gamma \vdash \texttt{if } C_1[e_1] \texttt{ then } C_2[e_1] \texttt{ else } C_3[e_1] : \tau']\!] \circ r} \\ = & [\![t_{\mathrm{true}} + t]\!] \circ \mathcal{V}[\![\Gamma \vdash C_2[e_1] : \tau']\!] \circ r \\ \preceq^{\mathrm{V}}_{\tau'} & [\![t_{\mathrm{true}} + t]\!] \circ \mathcal{V}[\![\Gamma \vdash C_2[e_2] : \tau']\!] \circ r \\ \preceq^{\mathrm{V}}_{\tau'} & [\![t_{\mathrm{true}} + t']\!] \circ \mathcal{V}[\![\Gamma \vdash C_2[e_2] : \tau']\!] \circ r \\ = & \mathcal{V}[\![\Gamma \vdash \texttt{if } C_1[e_2] \texttt{ then } C_2[e_2] \texttt{ else } C_3[e_2] : \tau']\!] \circ r \end{array}$$

Similarly if b is ff, then

$$\begin{array}{cl} \multicolumn{2}{l}{\mathcal{V}[\![\Gamma \vdash \texttt{if } C_1[e_1] \texttt{ then } C_2[e_1] \texttt{ else } C_3[e_1] : \tau']\!] \circ r} \\ = & [\![t_{\mathrm{false}} + t]\!] \circ \mathcal{V}[\![\Gamma \vdash C_3[e_1] : \tau']\!] \circ r \\ \preceq^{\mathrm{V}}_{\tau'} & [\![t_{\mathrm{true}} + t']\!] \circ \mathcal{V}[\![\Gamma \vdash C_3[e_2] : \tau']\!] \circ r \\ = & \mathcal{V}[\![\Gamma \vdash \texttt{if } C_1[e_2] \texttt{ then } C_2[e_2] \texttt{ else } C_3[e_2] : \tau']\!] \circ r \end{array}$$

$C[\,] = \texttt{rec } x.C'[\,]$

Let $f_0 \leq f_1 \leq \ldots : \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!] \to C\mathcal{T}^{\mathrm{V}}[\![\tau']\!]$ be a sequence where $f_0 = \bot \circ !$ and

$$f_{n+1} = [\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau' \vdash C'[e_1] : \tau']\!] \circ \langle \mathsf{id}, f_n \rangle$$

Then

$$\mathcal{V}[\![\texttt{rec } x.C'[e_1]]\!]\tau' \circ r = \mathsf{fixp}([\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau' \vdash C'[e_1] : \tau']\!]) \circ r = (\bigsqcup_{n=0}^{\infty} f_n) \circ r = \bigsqcup_{n=0}^{\infty} (f_n \circ r)$$

Similarly, let $f'_0 \leq f'_1 \leq \ldots : \mathcal{T}^{\mathrm{V}}[\![\Gamma]\!] \to C\mathcal{T}^{\mathrm{V}}[\![\tau']\!]$ be a sequence where $f'_0 = \bot \circ !$ and $f'_{n+1} = [\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau' \vdash C'[e_2] : \tau']\!] \circ \langle \mathsf{id}, f'_n \rangle$. Then $\mathcal{V}[\![\texttt{rec } x.C'[e_2]]\!]\tau' \circ r = \bigsqcup_{n=0}^{\infty}(f'_n \circ r)$. Now $f_0 \circ r \preceq^{\mathrm{V}}_{\tau'} f'_0 \circ r$ because both are $\bot$. If $f_n \circ r \preceq^{\mathrm{V}}_{\tau'} f'_n \circ r$, then, by the induction hypothesis,

$$\begin{array}{rcl} f_{n+1} \circ r & = & [\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau' \vdash C'[e_1] : \tau']\!] \circ \langle \mathsf{id}, f_n \rangle \circ r \\ & = & [\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau' \vdash C'[e_1] : \tau']\!] \circ \langle r, f_n \circ r \rangle \\ & \preceq^{\mathrm{V}}_{\tau} & [\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau' \vdash C'[e_1] : \tau']\!] \circ \langle r, f'_n \circ r \rangle \\ & \preceq^{\mathrm{V}}_{\tau} & [\![t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma, x : \tau' \vdash C'[e_2] : \tau']\!] \circ \langle r, f'_n \circ r \rangle \\ & = & f'_{n+1} \circ r \end{array}$$

Thus $\bigsqcup_{n=0}^{\infty}(f_n \circ r) \preceq^{\mathrm{V}}_{\tau} \bigsqcup_{n=0}^{\infty}(f'_n \circ r)$, so $\mathcal{V}[\![\Gamma \vdash C[e_1] : \tau']\!] \preceq^{\mathrm{V}}_{\tau} \mathcal{V}[\![\Gamma \vdash C[e_2] : \tau']\!]$.

□

To prove soundness of $\preceq^{\mathrm{V}}_{\tau}$, we need to make one assumption about the ground type values: that $\mathcal{V}[\![v]\!] = \mathcal{V}[\![v']\!]$ implies that $v = v'$. While this assumption is not reasonable for higher order types, it is expected for ground types. If it failed to hold for ground types there would be distinct values of ground type that are indistinguishable operationally. Also, if our extensional category is **PDom**, then we can safely set A_g to be the set of operational values of that type, discretely ordered, and thus guarantee that the property exists.

Therefore assume that $\mathcal{V}[\![e_1 : \tau]\!] \preceq^{\mathrm{V}}_{\tau} \mathcal{V}[\![e_2 : \tau]\!]$. Let $C[\,]$ be a context such that, for some ground type g, $C[e_1]$ is a closed expression of type g. Then $\vdash C[e_2] : g$ and $\mathcal{V}[\![C[e_1] : g]\!] \preceq^{\mathrm{V}}_{\tau} \mathcal{V}[\![C[e_2] : g]\!]$. If $\mathcal{V}[\![C[e_1] : g]\!] = \bot$, then by adequacy there cannot exist a value v or a cost t such that $C[e_1] \overset{t}{\Rightarrow}_{\mathrm{v}} v$. Similarly, $\mathcal{V}[\![C[e_2] : g]\!] = \bot$, so there cannot exist a value v or a cost t such that $C[e_1] \overset{t}{\Rightarrow}_{\mathrm{v}} v$. Therefore $C[e_1] \trianglelefteq^{\mathrm{V}} C[e_2]$. If $\mathcal{V}[\![C[e_1] : g]\!]$ is not $\bot$, then by adequacy there exists a cost t and a value v such that $C[e_1] \overset{t}{\Rightarrow}_{\mathrm{v}} v$, which means that $\mathcal{V}[\![C[e_1] : g]\!] = [\![t]\!] \circ \mathcal{V}[\![v : g]\!]$. Thus there exists a cost t' such that $t \preceq t'$ and $\mathcal{V}[\![C[e_2] : g]\!] = [\![t']\!] \circ \mathcal{V}[\![v : g]\!]$. By adequacy and the assumption made on values, we thus know that $C[e_2] \overset{t'}{\Rightarrow}_{\mathrm{v}} v$. Therefore $C[e_1] \trianglelefteq^{\mathrm{V}} C[e_2]$, so $e_1 \trianglelefteq^{\mathrm{V}} e_2$.

There is one more interesting property we can show, by using a particular cost order (and thus a particular improvement relation). Given two costs t_1, t_2, let $t_1 \overset{t}{\preceq} t_2$ if, for some non-negative integer m, $t_2 = nt + t_1$. If we assume that $+$ is commutative, then $\overset{t}{\preceq}$ is a cost order. It is a limited ordering, stating that the cost differs solely by a constant factor times a fixed cost t‘. If e is a closed expression of type τ, and if for some cost t, $e \overset{t}{\Rightarrow}_{\mathrm{v}} v$, then by soundness $\mathcal{V}[\![e : \tau]\!] = [\![t]\!] \circ \mathcal{V}[\![v : \tau]\!]$, so $\mathcal{V}[\![v : \tau]\!] \overset{t}{\preceq}_g \mathcal{V}[\![e : \tau]\!]$. Therefore, for any context $C[\,]$, if $\Gamma \vdash C[e] : \tau$, then $\mathcal{V}[\![\Gamma \vdash C[v] : \tau]\!] \overset{t}{\preceq}_{\tau} \mathcal{V}[\![\Gamma \vdash C[e] : \tau]\!]$, as well. This means that replacing an expression with its value can only affect the cost by some constant factor of t. The exact factor will vary depending on how many times e is actually evaluated; while e is only evaluated once when applied to a function, when filling a hole it may be evaluated several times. For example let $C[\,] = \mathtt{twice}(\mathtt{lam}\ x.[\,])(\overline{3})$. Then for any expression e (such that e is well typed when x has type **nat**), $C[e]$ evaluates e twice. This result is true regardless of the choice of cost order; it is the choice of cost order that made the result readily apparent.

3.10 Conclusion

In this chapter we developed semantic models for the call-by-value semantics. These models include the evaluation time as well the final result. The definition of "time" came from the operational semantics; we counted the number of occurrences of operations such as applying abstractions, unrolling recursions, taking a conditional, or adding two numbers. This definition of cost is similar to, but more flexible than, the definition of cost used in [37], which counted the depth of the tree generated by the evaluation. It is also similar to [50] which counts elementary operations such as application of a function or equality tests. By setting some of the cost constants (such as t_{true}) to 0 we obtain the definition of cost used in ([6], [22])) which counted the number of function calls or recursion unrollings. We can also emulate the definition used in ([12], [41]) which only counted cost in the primitive functions. Therefore our definition has an advantage: it is highly flexible, while still returning the same general costs as in other research.

The original choice of operational semantics has a significant impact on the definition of "time"; because the operational semantics is relatively abstract, costs are related to abstract operations rather than memory accesses or stack manipulations. Also, because the operational semantics uses substitution for application, values necessarily have 0 cost, and we cannot, in general, distinguish between the application of different abstractions; both restrictions are necessary in order for the Substitution Lemma to hold. We also cannot assign costs to the evaluation of variables because free variables never occur in the operational semantics. In most cases the cost normally attributed to constructors is directly proportional to the number of applications or recursive calls in a function, so the overall complexity is the same even without the the cost of forming values. Alternatively, we can construct additional constants as described in section 3.2. This construction has the effect of separating the *construction* of values from the *use* of already constructed values, which in many cases can be a useful distinction.

For complexity analysis, it is usually not necessary to distinguish the cost of applying different functions. If there is a need to do so, however, a less abstract operational model will be required.

In Sections 3.3 through 3.3.1 we developed a denotational semantics including cost. The compositionality inherent in the denotational semantics makes it easier to compute and compare costs of expressions as we do not need to be concerned with external effects. We are also better able to examine the effects of non-termination, which becomes more important for the call-by-name and call-by-need semantics.

For the denotational semantics we created a mathematical structure in category theory called a cost structure. Cost structures resemble strong monads in structure and except for additional elements governing the additions and removal of costs. We do not need all the properties required for strong monads in order to form a sound and adequate semantics, although we call a cost structure that has those properties a strong cost structure. The extra properties are not needed because, as indicated by adequacy, we are only concerned with elements of the form $\perp$ or $[\![t]\!] \circ \eta \circ y$, for $y : \mathbf{1} \to A$.

In defining the structure $(\mathbf{T}, \eta^{\mathrm{T}}, (-)^{\star}, \psi^{\mathrm{T}}, [\![-]\!]^{\mathrm{T}}, \delta^{\mathrm{T}}, \mathsf{reduce})$ we showed that non-trivial cost structures exist and furthermore that the non-trivial cost structure is also strong. In the process we defined arrow cost structures, which generate cost structures. While they have more elements, it is generally easier to define arrow cost structures than generic cost structure as the strictness and separability conditions are more easily proven.

The process of deriving a denotational semantics with cost structures is straightforward: We replace the lifting monad elements of the extensional semantics by the equivalent elements of the cost structure and add extra cost as specified by the operational semantics. The denotational semantics thus derived is sound, adequate and separable. We were, as a result, able to use the semantics derived to calculate the complexity of a number of examples. We were also able to define what it meant for a program to be “faster” than another program and that the derived ordering was compositional.

Chapter 4

Call-by-name

In this chapter we examine another form of evaluation strategy called *call-by-name*. When performing an application $e_1(e_2)$ under call-by-name the expression e_2 is substituted into the appropriate places in e_1 without any initial evaluation. Thus if e_1 never needs e_2, it is never evaluated; on the other hand, if e_1 needs e_2 multiple times, the expression is evaluated multiple times as well.

Because of the inefficiencies of re-evaluation there are not many languages implementations that use call-by-name directly. Some languages, such as C, have macro definitions which behave like call-by-name functions; in C, macros are used to avoid the overhead of a function call. Macros generally are simple enough that the duplication of code is under control. Algol-60 is an example of a call-by-name language, and it has some language features that take advantage of that fact. It also has a call-by-value `let` construct to force evaluation and thus selectively protect against reevaluation; we will see in section 4.5 that without such a construct many simple programs would become substantially less efficient.

Frequently, instead of using call-by-name directly, a language will implement call-by-name functionality with the results stored for future use. This implementation is not safe for Algol-60 because of possible changes in state but is safe for any pure functional language such as Haskell and Miranda. Such languages are called *lazy* or *call-by-need* as opposed to call-by-name.

We include the study of the call-by-name strategy for three reasons: it provides a way to add non-strict elements without encountering the difficulties of the call-by-need semantics (where one must keep track of which expressions have already been evaluated); the call-by-name semantics has been frequently used in definitions of denotational semantics using categories and so is relatively familiar to many; and it demonstrates the generality of the cost structure because it can be used for the call-by-name semantics as well as the call-by-value. Note that for some programs we never re-evaluate an argument, so the cost computed for them should apply even when evaluating under a call-by-need strategy.

The general form of the language is the same as for call-by-value. One primary exception is the include of "lazy" constants; namely those that do not automatically evaluate all of their arguments. Therefore there are now two sets of constructors: the strict constructors, **S-Construct**, which behave the same as for the call-by-value, and lazy constructors, **L-Construct**, which do not evaluate their subparts. Constructors which evaluate some, but not all, of their subparts are not considered at this time though they can be simulated if necessary.

Because we now have non-strict constants, it is possible to define `if` as a constant rather than as a built-in statement. The gain in insight or simplicity, however, is not worth the loss in consistency so the `if` construct is still built into the call-by-name language.

The construction of both the denotational semantics and the operational semantics for the call-

$$c \overset{0}{\Rightarrow}_{\mathrm{n}} c \qquad\qquad \texttt{lam}\ x.e \overset{0}{\Rightarrow}_{\mathrm{n}} \texttt{lam}\ x.e$$

$$\frac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} ce'_1 \ldots e'_i \quad \mathrm{napply}(c, e'_1, \ldots, e'_i, e_2) \overset{t'}{\Rightarrow} v}{e_1(e_2) \overset{t'+t_1}{\Rightarrow}_{\mathrm{n}} v} \qquad \frac{[\texttt{rec}\ x.e/x]e \overset{t}{\Rightarrow}_{\mathrm{n}} v}{\texttt{rec}\ x.e \overset{t+t_{\mathrm{rec}}}{\Rightarrow}_{\mathrm{n}} v}$$

$$\frac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \texttt{lam}\ x.e' \quad [e_2/x]e' \overset{t_2}{\Rightarrow}_{\mathrm{n}} v}{e_1(e_2) \overset{t_2+t_{\mathrm{app}}+t_1}{\Rightarrow}_{\mathrm{n}} v}$$

$$\frac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \texttt{true} \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{n}} v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \overset{t_2+t_{\mathrm{true}}+t_1}{\Rightarrow}_{\mathrm{n}} v} \qquad \frac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \texttt{false} \quad e_3 \overset{t_3}{\Rightarrow}_{\mathrm{n}} v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \overset{t_3+t_{\mathrm{false}}+t_1}{\Rightarrow}_{\mathrm{n}} v}$$

Figure 4.1: Natural operational semantics for call-by-name functional language

$$\mathrm{apply}(ce_1, \ldots,, e_i) \overset{0}{\Rightarrow} ce_1 \ldots e_i \ \ (i < \mathrm{ar}(c))$$
$$\mathrm{apply}(ce_1, \ldots,, e_{\mathrm{ar}(c)}) \overset{0}{\Rightarrow} ce_1 \ldots e_{\mathrm{ar}(c)} \ \ (c \in \textbf{L-Construct})$$

$$\frac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} v_1 \ldots e_n \overset{t_n}{\Rightarrow}_{\mathrm{n}} v_n}{\mathrm{apply}(c, e_1, \ldots,, e_n) \overset{t_n+\ldots+t_1}{\Rightarrow} cv_1 \ldots v_n} (c \in \textbf{S-Construct}, n = \mathrm{ar}(c))$$

Figure 4.2: Known rules for constant application

by-name follows the same general pattern as it did for the call-by-value semantics. The operational semantics is, as before, derived from the extensional semantics given in Chapter 2 by labeling transitions with costs. Furthermore, the same cost structure used in the previous chapter to convert the extensional call-by-value denotational semantics to an intensional semantics is sufficient to convert the call-by-name semantics as well. Therefore in this chapter we repeat the soundness and adequacy proofs for call-by-name and discuss how the addition of lazy constructors changes the assumptions we must make about constants. These assumptions, however, still hold for the FL language, although we need to alter the meaning of some data types to allow laziness, particularly lists. We end the chapter by re-examining a number of the examples given in the previous chapter. From these examples we see how using call-by-name can produce a substantial change in the complexity of a program. The examples also allow us to explore ways to examine costs with lazy lists when the costs may be spread out throughout the list.

4.1 Operational semantics

The primary operational semantics are listed in Figure 4.1. These are identical to the operational semantics defined in chapter 2 except that the transitions have been labeled with costs. The restrictions on the choice of costs are the same as for the call-by-value semantics (e.g., values must have 0 cost); therefore, the labels are the same as for the call-by-value semantics, given the overall differences in the semantic rules themselves.

Some of the rules for constants, those that can be specified exactly, are listed in Figure 4.2. These include rules for constructors (of both types) and incomplete applications.

The intensional and extensional operational semantics are closely related, in that $e \Rightarrow_{\mathrm{n}} v$ if and

$$
\begin{array}{lcl}
\mathcal{N}[\![\Gamma \vdash c : \tau]\!] & = & \mathsf{raise}^{(\mathrm{ar}(c))}(\mathcal{C}^{\mathrm{N}}[\![c : \tau]\!]) \circ !_{\mathcal{T}^{\mathrm{N}}[\![\Gamma]\!]} \\
\mathcal{N}[\![\Gamma \vdash x_i : \tau_i]\!] & = & \pi_i^n \\
\mathcal{N}[\![\Gamma \vdash \mathtt{lam}\ x.e : \tau' \to \tau]\!] & = & \eta \circ \mathsf{curry}(\mathcal{N}[\![\Gamma, x : \tau' \vdash e : \tau]\!]) \\
\mathcal{N}[\![\Gamma \vdash e_1(e_2) : \tau]\!] & = & \mathsf{app}^* \circ \psi \circ \langle \mathcal{N}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!], \\
 & & \quad \eta \circ \mathcal{N}[\![\Gamma \vdash e_2 : \tau']\!]\rangle \\
\mathcal{N}[\![\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau]\!] & = & \mathsf{cond}^* \circ \psi \circ \langle \mathcal{N}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!], \\
 & & \quad \langle \mathcal{N}[\![\Gamma \vdash e_2 : \tau]\!], \mathcal{N}[\![\Gamma \vdash e_3 : \tau]\!]\rangle\rangle \\
\mathcal{N}[\![\Gamma \vdash \mathtt{rec}\ x.e : \tau]\!] & = & \mathsf{fixp}(\mathcal{N}[\![\Gamma, x : \tau \vdash e : \tau]\!]) \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{true} : \mathbf{bool}]\!] & = & \eta \circ \mathsf{tt} \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{false} : \mathbf{bool}]\!] & = & \eta \circ \mathsf{ff} \\
\mathsf{raise}(f) & = & \eta \circ \mathsf{curry}(f)
\end{array}
$$

Figure 4.3: The converted call-by-name semantics

only if for some cost t, $e \stackrel{t}{\Rightarrow}_{\mathrm{n}} v$. Thus we immediately know that the operational semantics is both type sound and value sound.

4.2 Denotational semantics

One advantage of the cost structure is that it can also be used for the call-by-name as well as the call-by-value semantics, and the intensional semantics can be derived from the extensional semantics using the same general method that we used to derive the call-by-value intensional semantics from the extensional semantics. Therefore assume that we have a cost structure $(C, \eta, (-)^*, \psi, [\![-]\!])$ on $\mathcal{C}^{\mathrm{I}}$ over $\mathcal{C}$. To form the call-by-name semantics, we take the extensional semantics from Figure 2.14, substitute the equivalent parts of the cost structure for the lifting monad, and then determine where we must add extra cost (such as t_{rec}). Figure 4.3 lists the intensional semantics before the extra costs are added. To see where to add the extra costs, note that, for the operational semantics, we added extra cost in the same or equivalent places as we did for the call-by-value operational semantics. Therefore it should not come as a surprise that extra cost is added to the denotational semantics in the same or equivalent locations as they were added in the call-by-value denotational semantics. The result is listed in Figure 4.4.

As for the meaning of types, we again assume that for each ground type g there exists an object A_g^{N} in $\mathcal{C}^{\mathrm{I}}$ such that $EA_g^{\mathrm{N}} = A_g$. In most cases $A_g^{\mathrm{N}} = A_g^{\mathrm{V}}$, but there are ground types (such as lazy integers) that contain internal costs; in those cases we would expect A_g^{N} to be different.

Similarly, assume that for each constructed type δ of arity n, there is an n-ary functor on $\mathcal{C}^{\mathrm{I}}$, F_δ^{N}, such that $EF_\delta^{\mathrm{N}} = F_\delta$. In some cases, such as products, F_δ^{N} is the same functor as F_δ^{V}. In other cases, such as lists, F_δ^{N} is different.

4.3 Soundness of the call-by-name semantics

The soundness proof for the call-by-name denotational semantics is very similar to the soundness proof for the call-by-value semantics. Other than some technical details concerning application, the primary difference involves the assumptions made about constants. In particular, the form of strict constructors differs significantly from that used in the call-by-value case.

$$
\begin{aligned}
\mathcal{T}^{\mathrm{N}}[\![g]\!] &= A_g^{\mathrm{N}} \\
\mathcal{T}^{\mathrm{N}}[\![\delta(\tau_1,\ldots,\tau_n)]\!] &= F_\delta^{\mathrm{N}}(C\mathcal{T}^{\mathrm{N}}[\![\tau_1]\!],\ldots,C\mathcal{T}^{\mathrm{N}}[\![\tau_n]\!]) \\
\mathcal{T}^{\mathrm{N}}[\![\tau_1 \to \tau_2]\!] &= [C\mathcal{T}^{\mathrm{N}}[\![\tau_1]\!] \Rightarrow C\mathcal{T}^{\mathrm{N}}[\![\tau_2]\!]] \\
\mathcal{T}^{\mathrm{N}}[\![x_1:\tau_1,\ldots,x_n:\tau_n]\!] &= \times(C\mathcal{T}^{\mathrm{N}}[\![\tau_1]\!],\ldots,C\mathcal{T}^{\mathrm{N}}[\![\tau_n]\!])
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{N}[\![\Gamma \vdash c:\tau]\!] &= \mathsf{raise}^{(\mathrm{ar}(c))}(\mathcal{C}^{\mathrm{N}}[\![c:\tau]\!]) \circ !_{\mathcal{T}^{\mathrm{N}}[\![\Gamma]\!]} \\
\mathcal{N}[\![\Gamma \vdash x_i:\tau_i]\!] &= \pi_i^n \\
\mathcal{N}[\![\Gamma \vdash \mathtt{lam}\ x.e:\tau' \to \tau]\!] &= \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![\Gamma, x:\tau' \vdash e:\tau]\!]) \\
\mathcal{N}[\![\Gamma \vdash e_1(e_2):\tau]\!] &= \mathsf{app}^* \circ \psi \circ \langle \mathcal{N}[\![\Gamma \vdash e_1:\tau' \to \tau]\!], \\
&\qquad \eta \circ \mathcal{N}[\![\Gamma \vdash e_2:\tau']\!]\rangle \\
\mathcal{N}[\![\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3:\tau]\!] &= \mathsf{cond}^* \circ \psi \circ \langle \mathcal{N}[\![\Gamma \vdash e_1:\mathbf{bool}]\!], \eta \circ \\
&\qquad \eta \circ \langle [\![t_{\mathrm{true}}]\!] \circ \mathcal{N}[\![\Gamma \vdash e_2:\tau]\!], \\
&\qquad [\![t_{\mathrm{false}}]\!] \circ \mathcal{N}[\![\Gamma \vdash e_3:\tau]\!]\rangle\rangle \\
\mathcal{N}[\![\Gamma \vdash \mathtt{rec}\ x.e:\tau]\!] &= \mathsf{fixp}([\![t_{\mathrm{rec}}]\!] \circ \mathcal{N}[\![\Gamma, x:\tau \vdash e:\tau]\!]) \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{true}:\mathbf{bool}]\!] &= \eta \circ \mathsf{tt} \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{false}:\mathbf{bool}]\!] &= \eta \circ \mathsf{ff} \\
\mathsf{raise}(f) &= \eta \circ \mathsf{curry}(f)
\end{aligned}
$$

Figure 4.4: Intensional denotational semantics for call-by-name

Therefore we must redefine soundness for constants:

Definition 4.3.1 For the call-by-name semantics, $\mathcal{C}^{\mathrm{N}}[\![c:\tau]\!]$ is sound at c if the following properties holds:

- If c has arity 0 or is a lazy constructor then $\mathcal{C}^{\mathrm{N}}[\![c:\tau]\!]$ factors through $\eta_{\mathcal{T}^{\mathrm{N}}[\![\tau]\!]}$.
- If n is the arity of c, so that for some types $\tau_1,\ldots,\tau_n,\tau'$, $\tau = \tau_1 \to \ldots \to \tau_n \to \tau'$, and if c is a strict constructor then there must exists a morphism $f : \times(\mathcal{T}^{\mathrm{N}}[\![\tau_1]\!],\ldots,\mathcal{T}^{\mathrm{N}}[\![\tau_n]\!]) \to \mathcal{T}^{\mathrm{N}}[\![\tau']\!]$ such that $\mathcal{C}^{\mathrm{N}}[\![c:\tau]\!] = (\eta \circ f)^* \circ \psi^{(n)}$.
- For all operational rules of the form

$$\frac{e'_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} v'_1 \ldots e'_k \overset{t_k}{\Rightarrow}_{\mathrm{n}} v'_k}{\mathrm{apply}\,(c, e_1, \ldots, e_n) \overset{t}{\Rightarrow} v}$$

where n is the arity of c, suppose that for all $1 \le i \le k$ and types τ'_i, $\vdash e'_i : \tau'_i$ implies that $\mathcal{N}[\![e'_i:\tau'_i]\!] = [\![t_i]\!] \circ \mathcal{N}[\![v'_i:\tau'_i]\!]$. Then if c has type $\tau_1 \to \ldots \to \tau_n \to \tau$ and if for each $1 \le j \le n$, e_i has type τ_i

$$\mathcal{C}^{\mathrm{N}}[\![c:\tau]\!] \circ \langle\rangle(\mathcal{N}[\![e_1:\tau_1]\!],\ldots,\mathcal{N}[\![e_n:\tau_n]\!]) = [\![t]\!] \circ \mathcal{N}[\![v:\tau]\!]$$

$\mathcal{C}^{\mathrm{N}}[\![-]\!]$ is *sound* if for all $c \in \mathbf{Const}_\tau$, $\mathcal{C}^{\mathrm{N}}[\![c:\tau]\!]$ is sound at c.

Note that strict constructors of arity greater than 0 do not factor through η. Strict constructors do not always form values; they do so only when their arguments are values themselves. In a later section, Lemma 4.3.6 shows that when their arguments are values, the total result does factor through η.

As before, the constants `true` and `false` are sound because their meanings factor through η.

4.3.1 Technical Lemmas

The switch and drop lemmas show how changes in the environment structure change the meaning of an expressions; in particular, it shows that the change is simply the addition of a product morphism that adjusts the inputs to match the new environment. The proofs of these two lemmas are very much the same as for the call-by-value semantics; we omit the proofs.

Lemma 4.3.1 (Switch Lemma) *Let*

$$\Gamma = x_1 : \tau_1, \ldots, x_{k-1} : \tau_{k-1}, x_k : \tau_k, \ldots, x_n : \tau_n$$

$$\Gamma' = x_1 : \tau_1, \ldots, x_k : \tau_k, x_{k-1} : \tau_{k-1}, \ldots, x_n : \tau_n$$

Then if $\Gamma \vdash e : \tau$,

$$\mathcal{N}[\![\Gamma \vdash e : \tau]\!] = \mathcal{N}[\![\Gamma' \vdash e : \tau]\!] \circ \beta^n_k$$

Lemma 4.3.2 (Drop Lemma) *If* $\Gamma \vdash e : \tau$ *and* x *is not free in* e*, then*

$$\mathcal{N}[\![\Gamma, x : \tau' \vdash e : \tau]\!] = \mathcal{N}[\![\Gamma \vdash e : \tau]\!] \circ \pi_1$$

The substitution lemma shows the relationship between substituting a variable in an expression and setting its value in the environment. It's proof is also substantially the same as the call-by-value version.

Lemma 4.3.3 (Substitution Lemma) *Suppose that* $\Gamma, x : \tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$*. Then*

$$\mathcal{N}[\![\Gamma \vdash [e'/x]e : \tau]\!] = \mathcal{N}[\![\Gamma, x : \tau' \vdash e : \tau]\!] \circ \langle \mathsf{id}, \mathcal{N}[\![\Gamma \vdash e' : \tau']\!]\rangle$$

Proof. By induction on the structure of e. The abstraction case, which differs a bit from the call-by-value version of the proof, is included here. If $e = \mathtt{lam}\ y.e''$, where $\tau = \tau_1 \to \tau_2$, then we can assume that $y \neq x$ and y is not free in e'. Therefore

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{N}[\![\Gamma \vdash [e'/x]\mathtt{lam}\ y.e'' : \tau_1 \to \tau_2]\!]} \\
= & \mathcal{N}[\![\Gamma \vdash \mathtt{lam}\ y.[e'/x]e'' : \tau_1 \to \tau_2]\!] & \\
= & \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![\Gamma, y : \tau_1 \vdash [e'/x]e'' : \tau_2]\!]) & \\
= & \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![\Gamma, y : \tau_1, x : \tau' \vdash e'' : \tau_2]\!] & \text{Induction hypothesis} \\
 & \quad \circ \langle \mathsf{id}, \mathcal{N}[\![\Gamma, y : \tau_1 \vdash e' : \tau']\!]\rangle) & \\
= & \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![\Gamma, y : \tau_1, x : \tau' \vdash e'' : \tau_2]\!] & \text{Drop lemma} \\
 & \quad \circ \langle \mathsf{id}, \mathcal{N}[\![\Gamma \vdash e' : \tau]\!] \circ \pi_1\rangle) & \\
= & \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![\Gamma, x : \tau', y : \tau_1 \vdash e'' : \tau_2]\!] & \text{Switch lemma} \\
 & \quad \circ \beta \circ \langle \mathsf{id}, \mathcal{N}[\![\Gamma \vdash e' : \tau]\!] \circ \pi_1\rangle) & \\
= & \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![\Gamma, x : \tau', y : \tau_1 \vdash e'' : \tau_2]\!] & \\
 & \quad \circ (\langle \mathsf{id}, \mathcal{N}[\![\Gamma \vdash e' : \tau]\!]\rangle \times \mathsf{id})) & \\
= & \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![\Gamma, x : \tau', y : \tau_1 \vdash e'' : \tau_2]\!]) & \\
 & \quad \circ \langle \mathsf{id}, \mathcal{N}[\![\Gamma \vdash e' : \tau]\!]\rangle & \\
= & \mathcal{N}[\![\Gamma, x : \tau' \vdash e : \tau]\!] \circ \langle \mathsf{id}, \mathcal{N}[\![\Gamma \vdash e' : \tau]\!]\rangle &
\end{array}$$

$\square$

4.3.2 Constant application

For the call-by-value semantics, a constant of arity greater than 0 denoted a function from values to values. Its action on possibly non-terminating input was uniform over all constants and thus we defined separately the meaning of constants on expressions. For the call-by-name language, however, such constants may denote a non-strict function and thus the rules for constants themselves must take into account non-terminating input. As a result, we give a general rule for $ce_1 \ldots e_n$ instead of $cv_1 \ldots v_n$ because we do not generally know the behavior of c on non-terminating inputs.

Lemma 4.3.4 *Suppose that $c \in \mathbf{Const}_{\tau_1 \to \ldots \to \tau_n \to \tau}$ has arity n, that $0 \le i \le n$, and that for $1 \le j \le i$, $\Gamma \vdash e_j : \tau_j$. Then*

$$\mathcal{N}[\![\Gamma \vdash ce_1 \ldots e_i : \tau_{i+1} \to \ldots \to \tau_n \to \tau]\!] = \mathsf{raise}^{(n-i)}(\mathcal{C}^{\mathrm{N}}[\![c]\!]) \circ \langle\rangle(\mathcal{N}[\![\Gamma \vdash e_1 : \tau_1]\!], \ldots, \mathcal{N}[\![\Gamma \vdash e_i : \tau_i]\!])$$

Proof. By induction on i. Let $\tau' = \tau_{i+1} \to \ldots \to \tau_n \to \tau$.

First, if $i = 0$ then

$$\begin{aligned} \mathcal{N}[\![\Gamma \vdash c : \tau]\!] &= \mathsf{raise}^{(n)}(\mathcal{C}^{\mathrm{N}}[\![c : \tau]\!]) \circ !_{\mathcal{T}^{\mathrm{N}}[\![\Gamma]\!]} \\ &= \mathsf{raise}^{(n-0)}(\mathcal{C}^{\mathrm{N}}[\![c : \tau]\!]) \circ \langle\rangle() \end{aligned}$$

Otherwise, if $i > 0$ then we can assume the lemma holds for $i - 1$. Therefore

$$\begin{aligned} &\mathcal{N}[\![\Gamma \vdash ce_1 \ldots e_i : \tau']\!] \\ &\quad= \mathcal{N}[\![\Gamma \vdash (ce_1 \ldots e_{i-1})(e_i) : \tau']\!] \\ &\quad= \mathsf{app}^* \circ \psi \circ \langle \mathcal{N}[\![\Gamma \vdash ce_1 \ldots e_{i-1} : \tau_i \to \tau']\!], \eta \circ \mathcal{N}[\![\Gamma \vdash e_i : \tau_i]\!]\rangle \\ &\quad= \mathsf{app}^* \circ \psi \circ \langle \mathsf{raise}^{(n-(i-1))}(\mathcal{C}^{\mathrm{N}}[\![c]\!]) && \text{Induction hypothesis} \\ &\qquad\qquad \circ \langle\rangle(\mathcal{N}[\![\Gamma \vdash e_1 : \tau_1]\!], \ldots, \mathcal{N}[\![\Gamma \vdash e_{i-1} : \tau_{i-1}]\!]), \eta \circ \mathcal{N}[\![\Gamma \vdash e_i : \tau_i]\!]\rangle \\ &\quad= \mathsf{app}^* \circ \psi \circ \langle \eta \circ \mathsf{curry}(\mathsf{raise}^{(n-i)}(\mathcal{C}^{\mathrm{N}}[\![c]\!])) \\ &\qquad\qquad \circ \langle\rangle(\mathcal{N}[\![\Gamma \vdash e_1 : \tau_1]\!], \ldots, \mathcal{N}[\![\Gamma \vdash e_{i-1} : \tau_{i-1}]\!]), \eta \circ \mathcal{N}[\![\Gamma \vdash e_i : \tau_i]\!]\rangle \\ &\quad= \mathsf{raise}^{(n-i)}(\mathcal{C}^{\mathrm{N}}[\![c]\!]) \circ \langle \langle\rangle(\mathcal{N}[\![\Gamma \vdash e_1 : \tau_1]\!], \ldots, \mathcal{N}[\![\Gamma \vdash e_{i-1} : \tau_{i-1}]\!]), && \text{Lemma 3.4.3} \\ &\qquad\qquad \mathcal{N}[\![\Gamma \vdash e_i : \tau_i]\!]\rangle \\ &\quad= \mathsf{raise}^{(n-i)}(\mathcal{C}^{\mathrm{N}}[\![c]\!]) \circ \langle\rangle(\mathcal{N}[\![\Gamma \vdash e_1 : \tau_1]\!], \ldots, \mathcal{N}[\![\Gamma \vdash e_i : \tau_i]\!]) \end{aligned}$$

□

4.3.3 Values

The changes to values and constants slightly alter the proof that the meaning of a value factors through η. Again the only assumptions required are that constructors and constants of arity 0 are sound, making this theorem available when proving that other constants are sound.

Because the $\psi^{(n)}$ morphisms are used critically for strict constructors there is one additional lemma needed for the proof, a generalization of the soundness property concerning ψ. This is a straightforward generalization - just as ψ combines costs (reversing the order) of its inputs, so does $\psi^{(n)}$.

Lemma 4.3.5 *For any $n \ge 0$, if for $1 \le i \le n$, $f_i : D \to D_i$, then for any $t_1, \ldots, t_n \in T$,*

$$\psi^{(n)} \circ \langle\rangle([\![t_1]\!] \circ \eta \circ f_1, \ldots, [\![t_n]\!] \circ \eta \circ f_n) = [\![t_n + \ldots + t_1]\!] \circ \eta \circ \langle\rangle(f_1, \ldots, f_n)$$

Proof. Straightforward by induction on n. □

Lemma 4.3.6 *Suppose that for any $c \in \mathbf{Const}_\tau$ that is either a constructor or has arity 0, $\mathcal{C}^{\mathrm{N}}[\![c : \tau]\!]$ is sound for c. Then for any value $\Gamma \vdash v : \tau$, $\mathcal{N}[\![\Gamma \vdash v : \tau]\!]$ factors through η.*
Proof. By induction on the derivation of the proof that v is a value. All of the cases except the one where $v = cv_1 \dots v_{\mathrm{ar}(c)}$ and c is a strict constructor have the same proof as in the call-by-value version. The exception is proven below.

Suppose that $v = cv_1 \dots v_n$ where n is the arity of c and c is a strict constructor. As $\Gamma \vdash v : \tau$, there exist types $\tau_1, \dots, \tau_n$ such that $c \in \mathbf{Construct}_{\tau_1 \to \dots \to \tau_n \to \tau}$ and for each $1 \leq i \leq n$, $\Gamma \vdash v_i : \tau_i$. Also because v is a value each v_i must be a value as well so by the induction hypothesis there exist morphisms f_i such that $\mathcal{N}[\![\Gamma \vdash v_i : \tau_i]\!] = \eta \circ f_i$. Furthermore c is a strict constructor, so there exists f' such that $\mathcal{C}^{\mathrm{N}}[\![c]\!] = (\eta \circ f')^* \circ \psi^{(n)}$. Thus

$$\begin{array}{rll}
\multicolumn{3}{l}{\mathcal{N}[\![\Gamma \vdash cv_1 \dots v_n : \tau]\!]} \\
= & \mathcal{C}^{\mathrm{N}}[\![c]\!] \circ \langle\rangle(\mathcal{N}[\![\Gamma \vdash v_1 : \tau_1]\!] \dots \mathcal{N}[\![\Gamma \vdash v_n : \tau_n]\!]) & \text{Lemma 4.3.4} \\
= & (\eta \circ f')^* \circ \psi^{(n)} \circ \langle\rangle(\eta \circ f_1, \dots, \eta \circ f_n) & \\
= & (\eta \circ f')^* \circ \eta \circ \langle\rangle(f_1, \dots, f_n) & \text{Lemma 4.3.5} \\
= & \eta \circ f' \circ \langle\rangle(f_1, \dots, f_n) &
\end{array}$$

showing that the meaning of v factors through η, as required. □

4.3.4 Soundness

Theorem 4.3.7 *Suppose $\vdash e : \tau$ and $e \overset{t}{\Rightarrow}_{\mathrm{n}} v$. Then*

$$\mathcal{N}[\![e : \tau]\!] = [\![t]\!] \circ \mathcal{N}[\![v : \tau]\!]$$

Proof. By induction on the structure of the derivation of $e \overset{t}{\Rightarrow}_{\mathrm{n}} v$. The only cases that are significantly different from the proof for the call-by-value cases are the ones involving application. These are shown below.

Case $\dfrac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \mathtt{lam}\ x.e' \quad [e_2/x]e' \overset{t_2}{\Rightarrow}_{\mathrm{n}} v}{e_1(e_2) \overset{t_2 + t_{\mathrm{app}} + t_1}{\Rightarrow}_{\mathrm{n}} v}$:

Because $\vdash e_1(e_2) : \tau$, there exists a type τ' such that $\vdash e_1 : \tau' \to \tau$ and $\vdash e_2 : \tau'$.

By the induction hypothesis

$$\begin{array}{rcl}
\mathcal{N}[\![e_1 : \tau' \to \tau]\!] & = & [\![t_1]\!] \circ \mathcal{N}[\![\mathtt{lam}\ x.e' : \tau' \to \tau]\!] \\
& = & [\![t_1]\!] \circ \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![x : \tau' \vdash e' : \tau]\!])
\end{array}$$

and

$$\mathcal{N}[\![[e_2/x]e' : \tau]\!] \quad = \quad [\![t_2]\!] \circ \mathcal{N}[\![v : \tau]\!]$$

Therefore

$$\begin{array}{rll}
\multicolumn{3}{l}{\mathcal{N}[\![e_1(e_2) : \tau]\!]} \\
= & \mathsf{app}^* \circ \psi \circ \langle \mathcal{N}[\![e_1 : \tau' \to \tau]\!], \eta \circ \mathcal{N}[\![e_2 : \tau']\!] \rangle & \\
= & \mathsf{app}^* \circ \psi \circ \langle [\![t_1]\!] \circ \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![x : \tau' \vdash e' : \tau]\!]), & \text{as noted} \\
& \quad \eta \circ \mathcal{N}[\![e_2 : \tau']\!] \rangle & \\
= & [\![t_1]\!] \circ [\![t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![x : \tau' \vdash e' : \tau]\!] \circ \langle \mathsf{id}, \mathcal{N}[\![e_2 : \tau']\!] \rangle & \text{Lemma 3.4.3} \\
= & [\![t_1]\!] \circ [\![t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![[e_2/x]e' : \tau]\!] & \text{substitution lemma} \\
= & [\![t_1]\!] \circ [\![t_{\mathrm{app}}]\!] \circ [\![t_2]\!] \circ \mathcal{N}[\![v : \tau]\!] & \\
= & [\![t_2 + t_{\mathrm{app}} + t_1]\!] \circ \mathcal{N}[\![v : \tau]\!] &
\end{array}$$

Case $\dfrac{e_1 \overset{t_1}{\Rightarrow}_{\rm n} ce'_1 \ldots e'_i \quad \text{napply}(c, e'_1, \ldots, e'_i, e_2) \overset{t'}{\Rightarrow} v}{e_1(e_2) \overset{t'+t_1}{\Rightarrow}_{\rm n} v}$:

Because $\vdash e_1(e_2) : \tau$, there exists a type τ' such that $\vdash e_1 : \tau' \to \tau$ and $\vdash e_2 : \tau'$. Therefore by type soundness $ce'_1 \ldots, e'_i$ has type $\tau' \to \tau$ and there exist types $\tau_1, \ldots, \tau_i$, such that c has type $\tau_1 \to \ldots \to \tau_i \to \tau' \to \tau$ and for each $1 \leq j \leq i$, e'_j is of type τ_j.

By the induction hypothesis, $\mathcal{N}[\![e_1 : \tau' \to \tau]\!] = [\![t_1]\!] \circ \mathcal{N}[\![ce'_1 \ldots e'_i : \tau' \to \tau]\!]$. Furthermore, since $ce'_1 \ldots e'_i$ is a value, there exists a morphism f such that $\mathcal{N}[\![ce'_1 \ldots e'_i : \tau' \to \tau]\!] = \eta \circ f$.

Next note that

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{N}[\![e_1(e_2) : \tau]\!]} \\
 & = \mathsf{app}^* \circ \psi \circ \langle \mathcal{N}[\![e_1 : \tau' \to \tau]\!], \eta \circ \mathcal{N}[\![e_2 : \tau']\!]\rangle & \\
 & = \mathsf{app}^* \circ \psi \circ \langle [\![t_1]\!] \circ \mathcal{N}[\![ce'_1 \ldots e'_i : \tau' \to \tau]\!], \eta \circ \mathcal{N}[\![e_2 : \tau']\!]\rangle & \text{induction hypothesis} \\
 & = \mathsf{app}^* \circ \psi \circ \langle [\![t_1]\!] \circ \eta \circ f, \eta \circ \mathcal{N}[\![e_2 : \tau']\!]\rangle & \text{Lemma 4.3.6} \\
 & = [\![t_1]\!] \circ \mathsf{app} \circ \langle f, \mathcal{N}[\![e_2 : \tau']\!]\rangle & \text{Lemma 3.4.3} \\
 & = [\![t_1]\!] \circ \mathsf{app}^* \circ \psi \circ \langle \eta \circ f, \eta \circ \mathcal{N}[\![e_2 : \tau']\!]\rangle & \text{Lemma 3.4.3 (again)} \\
 & = [\![t_1]\!] \circ \mathsf{app}^* \circ \psi \circ \langle \mathcal{N}[\![ce'_1 \ldots e'_i : \tau' \to \tau]\!], \eta \circ \mathcal{N}[\![e_2 : \tau']\!]\rangle & \\
 & = [\![t_1]\!] \circ \mathcal{N}[\![ce'_1 \ldots e'_i e_2]\!] &
\end{array}$$

Let n be the arity of c. There are four possible ways to derive $\text{apply}(c, e'_1, \ldots, e'_i, e_2) \overset{t'}{\Rightarrow} v$:

- $i < n-1$: Then $v = ce'_1 \ldots e'_i e_2$ and $t' = 0$, so

$$\begin{aligned}
\mathcal{N}[\![e_1(e_2) : \tau]\!] &= [\![t_1]\!] \circ \mathcal{N}[\![ce'_1 \ldots e'_i e_2 : \tau]\!] \\
&= [\![t' + t_1]\!] \circ \mathcal{N}[\![v : \tau]\!]
\end{aligned}$$

- $i = n-1$ and c is a lazy constructor: As above, $v = ce'_1 \ldots e_i v_2$ and $t' = 0$, so the proof is the same as just shown.

- $i = n-1$ and c is a strict constructor: Then the derivation of $\text{napply}(c, e'_1, \ldots, e'_i, e_2) \overset{t'}{\Rightarrow} v$ has the following form:

$$\frac{e'_1 \overset{t'_1}{\Rightarrow}_{\rm n} v'_1 \ldots e'_i \overset{t'_i}{\Rightarrow}_{\rm n} v'_i \qquad e_2 \overset{t_2}{\Rightarrow}_{\rm n} v_2}{\text{napply}(ce'_1 \ldots e'_{n-1}, e'_i) \overset{t'}{\Rightarrow} cv'_1 \ldots v'_i v_2}$$

  where $t' = t_2 + t'_i + \ldots + t'_1$. By the induction hypothesis $\mathcal{N}[\![e'_j : \tau_j]\!] = [\![t'_j]\!] \circ \mathcal{N}[\![v'_j : \tau_j]\!]$ for each $1 \leq j \leq i$ and $\mathcal{N}[\![e_2 : \tau']\!] = [\![t_2]\!] \circ \mathcal{N}[\![v_2 : \tau']\!]$. Because each v'_j is a value, there must also exist morphisms f'_j such that for each $1 \leq j \leq i$, $\mathcal{N}[\![v'_j : \tau_j]\!] = \eta \circ f'_j$. Similarly there must exist a morphism f_2 such that $\mathcal{N}[\![v_2 : \tau']\!] = \eta \circ f_2$. Finally c is a strict constructor, so there exists a morphism f' such that $\mathcal{C}^{\rm N}[\![c]\!] = (\eta \circ f')^* \circ \psi^{(n)}$. Next, by the induction hypothesis and Lemma 4.3.5,

$$\begin{array}{ll}
\multicolumn{2}{l}{\psi^{(n)} \circ \langle\rangle(\mathcal{N}[\![e'_1 : \tau_1]\!], \ldots, \mathcal{N}[\![e'_i : \tau_i]\!], \mathcal{N}[\![e_2 : \tau']\!])} \\
 & = \psi^{(n)} \circ \langle\rangle([\![t'_1]\!] \circ \mathcal{N}[\![v'_1 : \tau_1]\!], \ldots, [\![t'_i]\!] \circ \mathcal{N}[\![v'_i : \tau_i]\!], [\![t_2]\!] \circ \mathcal{N}[\![v_2 : \tau_2]\!]) \\
 & = \psi^{(n)} \circ \langle\rangle([\![t'_1]\!] \circ \eta \circ f'_1, \ldots, [\![t'_i]\!] \circ \eta \circ f'_i, [\![t_2]\!] \circ \eta \circ f_2) \\
 & = [\![t_2 + t'_i + \ldots + t'_1]\!] \circ \eta \circ \langle\rangle(f'_1, \ldots, f'_i, f_2)
\end{array}$$

Therefore

$$
\begin{array}{lll}
\mathcal{N}[\![e_1(e_2) : \tau]\!] & & \\
\quad = & [\![t_1]\!] \circ \mathcal{N}[\![ce'_1 \ldots e'_i e_2 : \tau]\!] & \\
\quad = & [\![t_1]\!] \circ \mathcal{C}^{\mathrm{N}}[\![c]\!] \circ \langle\rangle(\mathcal{N}[\![e'_1 : \tau_1]\!], \ldots, \mathcal{N}[\![e'_i : \tau_i]\!], \mathcal{N}[\![e_2 : \tau']\!]) & \text{Lemma 4.3.4} \\
\quad = & [\![t_1]\!] \circ (\eta \circ f')^* \circ \psi^{(n)} \circ \langle\rangle(\mathcal{N}[\![e'_1 : \tau_1]\!], \ldots, \mathcal{N}[\![e'_i : \tau_i]\!], \mathcal{N}[\![e_2 : \tau']\!]) & \\
\quad = & [\![t_1]\!] \circ (\eta \circ f')^* \circ [\![t_2 + t'_i + \ldots + t'_1]\!] \circ \eta \circ \langle\rangle(f'_1, \ldots, f'_i, f_2) & \text{as noted} \\
\quad = & [\![t_1]\!] \circ [\![t_2 + t'_i + \ldots + t'_1]\!] \circ \eta \circ f' \circ \langle\rangle(f'_1, \ldots, f'_i, f_2) & \\
\quad = & [\![t_2 + t'_i + \ldots + t'_1 + t_1]\!] \circ (\eta \circ f')^* \circ \eta \circ \langle\rangle(f'_1, \ldots, f'_i, f_2) & \\
\quad = & [\![t' + t_1]\!] \circ (\eta \circ f')^* \circ \eta \circ \langle\rangle(f'_1, \ldots, f'_i, f_2) & \\
\quad = & [\![t' + t_1]\!] \circ (\eta \circ f')^* \circ \psi^{(n)} \circ \langle\rangle(\eta \circ f'_1, \ldots, \eta \circ f'_i, f_2) & \text{Lemma 4.3.5} \\
\quad = & [\![t' + t_1]\!] \circ \mathcal{C}^{\mathrm{N}}[\![c]\!] \circ \langle\rangle(\eta \circ f'_1, \ldots, \eta \circ f'_i, f_2) & \\
\quad = & [\![t' + t_1]\!] \circ \mathcal{C}^{\mathrm{N}}[\![c]\!] \circ \langle\rangle(\mathcal{N}[\![v'_1 : \tau_1]\!], \ldots, \mathcal{N}[\![v'_i : \tau_i]\!], \mathcal{N}[\![v_2 : \tau']\!]) & \\
\quad = & [\![t' + t_1]\!] \circ \mathcal{N}[\![cv'_1 \ldots v'_i, v_2]\!] &
\end{array}
$$

- $i = n - 1$ and c is not a constructor: Then by the soundness assumptions for constants and Lemma 4.3.4,

$$
\begin{array}{ll}
\mathcal{N}[\![ce'_1 \ldots e'_i e_2 : \tau]\!] & \\
\quad = & \mathsf{raise}^{(n-n)}(\mathcal{C}^{\mathrm{N}}[\![c]\!]) \circ \langle\rangle(\mathcal{N}[\![e'_1 : \tau_1]\!], \ldots, \mathcal{N}[\![e'_i : \tau_i]\!], \mathcal{N}[\![e_2 : \tau']\!]) \\
\quad = & \mathcal{C}^{\mathrm{N}}[\![c]\!] \circ \langle\rangle(\mathcal{N}[\![e'_1 : \tau_1]\!], \ldots, \mathcal{N}[\![e'_i : \tau_n]\!], \mathcal{N}[\![e_2 : \tau']\!]) \\
\quad = & [\![t']\!] \circ \mathcal{N}[\![v : \tau]\!]
\end{array}
$$

therefore

$$
\begin{array}{rcl}
\mathcal{N}[\![e_1(e_2) : \tau]\!] & = & [\![t_1]\!] \circ \mathcal{N}[\![ce'_1 \ldots e'_i e_2 : \tau]\!] \\
& = & [\![t_1]\!] \circ [\![t']\!] \circ \mathcal{N}[\![v : \tau]\!] \\
& = & [\![t' + t_1]\!] \circ \mathcal{N}[\![v : \tau]\!]
\end{array}
$$

$\square$

4.3.5 Soundness of the extensional semantics

The definition of the intensional semantics was designed so that $E\mathcal{N}[\![\Gamma \vdash e : \tau]\!] = \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!]$. By the intensional proof of soundness we know that if $e \stackrel{t}{\Rightarrow}_{\mathrm{n}} v$ and $\vdash e : \tau$ then $\mathcal{N}[\![e : \tau]\!] = [\![t]\!] \circ \mathcal{N}[\![v : \tau]\!]$. Therefore, as $E[\![t]\!] = \mathsf{id}$, $\mathcal{N}_{\mathrm{E}}[\![e : \tau]\!] = \mathcal{N}_{\mathrm{E}}[\![v : \tau]\!]$, so the extensional semantics is also sound.

4.3.6 Adequacy of the intensional semantics

The proof that the intensional semantics is adequate is essentially the same as it was for the call-by-value case because it depends primarily on properties of the cost structure.

Theorem 4.3.8 *For any closed expression e of type τ, $\mathcal{N}[\![e : \tau]\!] \neq \bot$ if and only if there exists a closed value v of type τ and a $t \in T$ such that $e \stackrel{t}{\Rightarrow}_{\mathrm{n}} v$.*

Proof. Suppose that $\mathcal{N}[\![e : \tau]\!] \neq \bot$. Then, by the properties of the cost structure, $\mathcal{N}_{\mathrm{E}}[\![e : \tau]\!]$, which equals $E\mathcal{N}[\![e : \tau]\!]$, is not $\bot$ either. By the adequacy of the extensional semantics there exists a closed value v of type τ such that $e \Rightarrow_{\mathrm{n}} v$. Therefore there exists a cost t such that $e \stackrel{t}{\Rightarrow}_{\mathrm{n}} v$.

Now suppose there exists a closed value v of type τ and a $t \in T$ such that $e \overset{t}{\Rightarrow}_{\mathrm{n}} v$. By Lemma 4.3.6 there exists a morphism $y : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}[\![\tau]\!]$ such that $\mathcal{N}[\![v : \tau]\!] = \eta \circ y$. Therefore, by soundness,

$$\mathcal{N}[\![e : \tau]\!] = [\![t]\!] \circ \mathcal{N}[\![v : \tau]\!] = [\![t]\!] \circ \eta \circ y \neq \bot$$

□

4.4 The language FL

At this point there have been a number of assumptions made about the behavior of constants without much of an attempt to show that the constants concerning products, etc., satisfy them. Given the differences in the formation of the constructed data types and projections it does not automatically follow that the assumptions are reasonable even though they proved to be in the call-by-value language.

Now that there are both lazy and strict constructors it becomes necessary to indicate which data types have which constructors. For a number of reasons, including some later examples, it will highly useful to have lazy lists, so let `cons` be a lazy constructor. Since the lists constants resemble product constants more than sum constants it will be more illuminating to let `pair` be strict and `inl` and `inr` be lazy. This choice also means that functions can return multiple values as a product, and we still know that the individual parts are also values.

4.4.1 Operational semantics

The operational semantics for the FL constants are listed in Figure 4.5. Again, we simply took the extensional operational semantics and added cost. For consistency, we chose the same additional costs for the call-by-name version as we did for the call-by-value. Thus most constants incur an additional cost independent of the arguments, with the exception of integer multiplication and integer equality test.

4.4.2 Denotational semantics

For the denotational semantics, it is possible to take advantage of a close relationship between the call-by-name and call-by-value for the integer constants, in particular, when

$$\text{vapply}\,(cv_1 \ldots v_{n-1}, v_n) \overset{t}{\Rightarrow} v$$

is a rule for the call-by-value semantics if and only if there is a rule of the form

$$\frac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} v_1 \ldots e_n \overset{t_n}{\Rightarrow}_{\mathrm{n}} v_n}{\text{napply}\,(ce_1 \ldots e_{n-1}, e_n) \overset{t+t_n+\ldots+t_1}{\Rightarrow} v}$$

for the call-by-name semantics. Note that this relationship does hold for constants such as $+$, but does not for list-related constants like `head`. Because $\mathcal{T}^{\mathrm{N}}[\![g]\!] = \mathcal{T}^{\mathrm{V}}[\![g]\!]$ for every ground type g of FL, the meaning of a call-by-name integer constant can be defined in terms of the call-by-value meanings. This way the soundness of the call-by-value constants can be used to prove the soundness of the call-by-name constants without having to look at each rule separately.

To see how to define $\mathcal{C}^{\mathrm{N}}[\![c]\!]$ in terms of $\mathcal{C}^{\mathrm{V}}[\![c]\!]$, first note that $\psi^{(n)} \circ \langle\rangle(f_1, \ldots, f_n) = \bot$ whenever at least one of the f_i's is $\bot$. Thus for any of these constants, if an argument fails to terminate and thus has a meaning of $\bot$, then the entire application has a meaning of $\bot$.

$$\dfrac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \overline{n} \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{n}} \overline{m}}{\mathrm{napply}(+, e_1, e_2) \overset{t_+ + t_2 + t_1}{\Rightarrow} \overline{n+m}} \qquad \dfrac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \overline{n} \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{n}} \overline{m}}{\mathrm{napply}(\times, e_1, e_2) \overset{t_*(n,m) + t_2 + t_1}{\Rightarrow} \overline{n*m}}$$

$$\dfrac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \overline{n} \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{n}} \overline{m}}{\mathrm{napply}(\dot{-}, e_1, e_2) \overset{t_- + t_2 + t_1}{\Rightarrow} \overline{0}}\ (n \le m) \qquad \dfrac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \overline{n} \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{n}} \overline{m}}{\mathrm{napply}(\dot{-}, e_1, e_2) \overset{t_- + t_2 + t_1}{\Rightarrow} \overline{n-m}}\ (n > m)$$

$$\dfrac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \overline{n} \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{n}} \overline{n}}{\mathrm{napply}(=, e_1, e_2) \overset{t_= + t_2 + t_1}{\Rightarrow} \mathtt{true}} \qquad \dfrac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \overline{n} \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{n}} \overline{m}}{\mathrm{napply}(=, e_1, e_2) \overset{t_{\neq} + t_2 + t_1}{\Rightarrow} \mathtt{false}}\ (n \neq m)$$

$$\dfrac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \overline{n} \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{n}} \overline{m}}{\mathrm{napply}(\le, e_1, e_2) \overset{t_{\le} + t_2 + t_1}{\Rightarrow} \mathtt{true}}\ (n \le m) \qquad \dfrac{e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} \overline{n} \quad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{n}} \overline{m}}{\mathrm{napply}(\le, e_1, e_2) \overset{t_{\le} + t_2 + t_1}{\Rightarrow} \mathtt{false}}\ (n > m)$$

$$\dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} <v_1, v_2>}{\mathrm{napply}(\mathtt{fst}, e) \overset{t + t_{\mathrm{fst}}}{\Rightarrow} v_1} \qquad \dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} <v_1, v_2>}{\mathrm{napply}(\mathtt{snd}, e) \overset{t + t_{\mathrm{snd}}}{\Rightarrow} v_2}$$

$$\dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} \mathtt{inl}(e') \quad e_1(e') \overset{t_1}{\Rightarrow}_{\mathrm{n}} v}{\mathrm{napply}(\mathtt{case}, e, e_1, e_2) \overset{t_1 + t + t_{\mathrm{case}}}{\Rightarrow} v} \qquad \dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} \mathtt{inr}(e') \quad e_2(e') \overset{t_2}{\Rightarrow}_{\mathrm{n}} v}{\mathrm{napply}(\mathtt{case}, e, e_1, e_2) \overset{t_2 + t + t_{\mathrm{case}}}{\Rightarrow} v}$$

$$\dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} e_1{::}e_2 \qquad e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} v}{\mathrm{napply}(\mathtt{head}, e) \overset{t_1 + t + t_{\mathrm{head}}}{\Rightarrow} v} \qquad \dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} e_1{::}e_2 \qquad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{n}} v}{\mathrm{napply}(\mathtt{tail}, e) \overset{t_2 + t + t_{\mathrm{tail}}}{\Rightarrow} v}$$

$$\dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} \mathtt{nil}}{\mathrm{napply}(\mathtt{nil?}, e) \overset{t + t_{\mathrm{nil?}}}{\Rightarrow} \mathtt{true}} \qquad \dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} e_1{::}e_2}{\mathrm{napply}(\mathtt{nil?}, e) \overset{t + t_{\mathrm{nil?}}}{\Rightarrow} \mathtt{false}}$$

Figure 4.5: Call-by-name application rules for FL

Next suppose that c is one of the integer constants, with arity n and $e_1, \ldots, e_n$ are closed expressions of the appropriate types (called $\tau_1, \ldots, \tau_n$) that also terminate. Then the meaning of each e_i can be assumed to be a morphism of the form $[\![t_i]\!] \circ \eta \circ f_i$, so

$$\begin{aligned}
&[\![t_n + \ldots + t_1]\!] \circ \mathcal{C}^{\mathrm{V}}[\![c]\!] \circ \langle\rangle(f_1, \ldots, f_n) \\
&\quad = \mathcal{C}^{\mathrm{V}}[\![c]\!]^* \circ [\![t_n + \ldots + t_1]\!] \circ \eta \circ \langle\rangle(f_1, \ldots, f_n) \\
&\quad = \mathcal{C}^{\mathrm{V}}[\![c]\!]^* \circ \psi^{(n)} \circ \langle\rangle([\![t_1]\!] \circ \eta \circ f_1, \ldots, [\![t_n]\!] \circ \eta \circ f_n) \\
&\quad = \mathcal{C}^{\mathrm{V}}[\![c]\!]^* \circ \psi^{(n)} \circ \langle\rangle(\mathcal{N}[\![\Gamma \vdash e_1 : \tau_1]\!], \ldots, \mathcal{N}[\![\Gamma \vdash e_n : \tau_n]\!])
\end{aligned}$$

Therefore it seems reasonable to set $\mathcal{C}^{\mathrm{N}}[\![c]\!]$ to $\mathcal{C}^{\mathrm{V}}[\![c]\!]^* \circ \psi^{(n)}$ where n is the arity of c. Intuitively, the meaning of c evaluates each argument, acts on the resulting values and adds in the cost of finding the values.

The meanings of the integer and boolean constants are listed in Figure 4.6. Remember that, for FL, we are operating in the category $\mathbf{PDom}^I$.

Products

For products, $F_\times^{\mathrm{N}}$ is the same as $F_\times^{\mathrm{V}}$, that is, it is the binary product functor $\mathcal{C}^{\mathrm{I}}$. This means that the meaning of $\tau_1 \times \tau_2$ is $C\mathcal{T}^{\mathrm{N}}[\![\tau_1]\!] \times C\mathcal{T}^{\mathrm{N}}[\![\tau_2]\!]$. Because we declared `pair` strict, the cost stored in

$$\begin{array}{rcl}
\mathcal{C}^{\mathrm{N}}[\![\overline{n}]\!] & = & (\eta \circ \mathsf{n}_n{}^I)^* \circ \psi^{(0)} = \eta \circ \mathsf{n}_n{}^I \\
\mathcal{C}^{\mathrm{N}}[\![=]\!] & = & (\mathsf{cond} \circ \langle \mathsf{id}, \langle [\![t_=]\!] \circ \eta, [\![t_{\neq}]\!] \circ \eta \rangle \rangle \circ \mathsf{eq}^I \circ (\pi_2 \times \mathsf{id}))^* \circ \psi^{(2)} \\
\mathcal{C}^{\mathrm{N}}[\![\leq]\!] & = & ([\![t_{\leq}]\!] \circ \eta \circ \mathsf{leq}^I \circ (\pi_2 \times \mathsf{id}))^* \circ \psi^{(2)} \\
\mathcal{C}^{\mathrm{N}}[\![+]\!] & = & ([\![t_+]\!] \circ \eta \circ \mathsf{plus}^I \circ (\pi_2 \times \mathsf{id}))^* \circ \psi^{(2)} \\
\mathcal{C}^{\mathrm{N}}[\![\dot{-}]\!] & = & ([\![t_-]\!] \circ \eta \circ \mathsf{minus}^I \circ (\pi_2 \times \mathsf{id}))^* \circ \psi^{(2)} \\
\mathcal{C}^{\mathrm{N}}[\![\times]\!] & = & ((\eta \circ \mathsf{times}^I)^* \circ \mathsf{imcost} \circ (\pi_2 \times \mathsf{id}))^* \circ \psi^{(2)}
\end{array}$$

Figure 4.6: Call-by-name semantics for integer and boolean constants in FL

$$\begin{array}{rcl}
\mathcal{C}^{\mathrm{N}}[\![\mathtt{pair}]\!] & = & (\eta \circ (\eta \times \eta) \circ (\pi_2 \times \mathsf{id}))^* \circ \psi^{(2)} \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{fst}]\!] & = & [\![t_{\mathrm{fst}}]\!] \circ \pi_1{}^* \circ \pi_2 \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{snd}]\!] & = & [\![t_{\mathrm{snd}}]\!] \circ \pi_2{}^* \circ \pi_2
\end{array}$$

Figure 4.7: Call-by-name semantics for product constants in FL

the product will always be 0; the internal cost is there solely to allow us a uniform definition for constants.

The extensional meaning of `pair` is $L(\mathsf{up} \times \mathsf{up}) \circ \mathsf{smash} \circ (\pi_2 \times \mathsf{id})$. By converting the lifting monad to the cost structure we get $(\eta \circ (\eta \times \eta))^* \circ \psi \circ (\pi_2 \times \mathsf{id})$. That equation, however, is not in the form required for sound strict constructors. Therefore we use some of the strong monad properties of lifting to convert the extensional meaning into the appropriate form:

$$\begin{array}{l}
L(\mathsf{up} \times \mathsf{up}) \circ \mathsf{smash} \circ (\pi_2 \times \mathsf{id}) \\
\qquad = \quad L(\mathsf{up} \times \mathsf{up}) \circ \mathsf{smash} \circ (L\pi_2 \times \mathsf{id}) \circ (\mathsf{smash} \times \mathsf{id}) \circ ((\mathsf{up} \times \mathsf{id}) \times \mathsf{id}) \\
\qquad = \quad L(\mathsf{up} \times \mathsf{up}) \circ L(\pi_2 \times \mathsf{id}) \circ \mathsf{smash} \circ (\mathsf{smash} \times \mathsf{id}) \circ ((\mathsf{up} \times \mathsf{id}) \times \mathsf{id}) \\
\qquad = \quad (\mathsf{up} \circ (\mathsf{up} \times \mathsf{up}) \circ (\pi_2 \times \mathsf{id}))^{\perp} \circ \mathsf{nsmash}(2)
\end{array}$$

Using the cost structure, this becomes

$$\mathcal{C}^{\mathrm{N}}[\![\mathtt{pair}]\!] = (\eta \circ (\eta \times \eta) \circ (\pi_2 \times \mathsf{id}))^* \circ \psi^{(2)}$$

which is of the required form while still ensuring that $E\mathcal{C}^{\mathrm{N}}[\![\mathtt{pair}]\!] = \mathcal{C}_{\mathrm{E}}^{\mathrm{N}}[\![\mathtt{pair}]\!]$.

The extensional meaning of `fst` is $\pi_1{}^{\perp} \circ \pi_2$. Converting to the cost structure gives us $\pi_1{}^* \circ \pi_2$. To that we add the cost of taking the first of a pair, t_{fst}, so

$$\mathcal{C}^{\mathrm{N}}[\![\mathtt{fst}]\!] = [\![t_{\mathrm{fst}}]\!] \circ \pi_1{}^* \circ \pi_2$$

Similarly, $\mathcal{C}^{\mathrm{N}}[\![\mathtt{fst}]\!] = [\![t_{\mathrm{snd}}]\!] \circ \pi_2{}^* \circ \pi_2$. By constructing their meanings from the extensional meanings we know that $E\mathcal{C}^{\mathrm{N}}[\![\mathtt{fst}]\!] = \mathcal{C}_{\mathrm{E}}^{\mathrm{N}}[\![\mathtt{fst}]\!]$ and $E\mathcal{C}^{\mathrm{N}}[\![\mathtt{snd}]\!] = E\mathcal{C}_{\mathrm{E}}^{\mathrm{N}}[\![\mathtt{snd}]\!]$.

Sums

As for products, the functor for sums is the same as for the call-by-value case, thus the meaning of $\tau_1 + \tau_2$ is $C\mathcal{T}^{\mathrm{N}}[\![\tau_1]\!] + C\mathcal{T}^{\mathrm{N}}[\![\tau_2]\!]$. The extensional meaning of `inl` is $\mathsf{up} \circ \mathsf{inl} \circ \pi_2$; therefore the

$$\begin{array}{rcl}
\mathcal{C}^{\mathrm{N}}[\![\mathtt{inl}]\!] & = & \eta \circ \mathsf{inl} \circ \pi_2 \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{inr}]\!] & = & \eta \circ \mathsf{inr} \circ \pi_2 \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{case}]\!] & = & [\![t_{\mathrm{case}}]\!] \circ (\mathsf{case}(\mathsf{nleft}^{\mathrm{I}}, \mathsf{nright}^{\mathrm{I}}))^* \circ \psi \circ \circ(\pi_2 \times \eta) \circ \alpha^r \\
\mathsf{nleft}^{\mathrm{I}} & = & \mathsf{app}^* \circ \psi \circ \beta \circ (\eta \times \pi_1) \\
\mathsf{nright}^{\mathrm{I}} & = & \mathsf{app}^* \circ \psi \circ \beta \circ (\eta \times \pi_2)
\end{array}$$

Figure 4.8: Call-by-name semantics for sum constants in FL

intensional meaning should be $\eta \circ \mathsf{inl} \circ \pi_2$ (using the definition of inl in $\mathbf{PDom}^I$). Similarly the meaning of `inr` is $\eta \circ \mathsf{inr} \circ \pi_2$. Both meanings have the form required of sound lazy constructors.

For `case`, its extensional meaning is

$$(\mathsf{case}(\mathsf{nleft}, \mathsf{nright}))^{\perp} \circ \mathsf{smash} \circ (\pi_2 \times \mathsf{up}) \circ \alpha^r$$

where

$$\begin{array}{rl}
\mathsf{nleft} = & \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \beta \circ (\mathsf{up} \times \pi_1) \\
\mathsf{nright} = & \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \beta \circ (\mathsf{up} \times \pi_2)
\end{array}$$

Converting to the cost structure we get

$$\begin{array}{rl}
\mathcal{C}^{\mathrm{N}}[\![\mathtt{case}]\!] = & (\mathsf{case}(\mathsf{nleft}^{\mathrm{I}}, \mathsf{nright}^{\mathrm{I}}))^* \circ \psi \circ (\pi_2 \times \eta) \circ \alpha^r \\
\mathsf{nleft}^{\mathrm{I}} = & \mathsf{app}^* \circ \psi \circ \beta \circ (\eta \times \pi_1) \\
\mathsf{nright}^{\mathrm{I}} = & \mathsf{app}^* \circ \psi \circ \beta \circ (\eta \times \pi_2)
\end{array}$$

we then add cost t_{case} to the beginning to get the result in Figure 4.8.

Lists

As mentioned, the functor $F^{\mathrm{N}}_{\mathbf{list}}$ is not the same as $F^{\mathrm{V}}_{\mathbf{list}}$. Also, because the tail of a list contains cost, $F^{\mathrm{N}}_{\mathbf{list}}$ is also not the lazy list functor Llist lifted to the arrow category.

Because we are using the arrow category $\mathbf{PDom}^I$, we can assume that there exists an arrow cost structure $(A, \eta^A, (-)^\star, \psi^A, [\![-]\!]^A, \delta, \mathsf{rd})$ from which we created $(C, \eta, (-)^*, \psi, [\![-]\!])$. Therefore, in $\mathbf{PDom}$, let $\mathsf{Alist}\,(X)$ be a domain satisfying the isomorphism

$$\mathsf{Alist}\,(X) \approx \mathbf{1} + (X \times LA\mathsf{Alist}\,(X))$$

with afold_X and $\mathsf{aunfold}_X$ as the isomorphisms. Also, given a morphism $f : X \to X'$, let the morphism $\mathsf{Alist}\,(f) : \mathsf{Alist}\,(X) \to \mathsf{Alist}\,(X')$ be the (least) solution to the equation

$$\mathsf{Alist}\,(f) = \mathsf{afold}_{X'} \circ (\mathsf{id} + (f \times L(\eta^A \circ \mathsf{Alist}\,(f))^\star)) \circ \mathsf{aunfold}_X$$

Unlike the extensional cases, we do not necessarily know via domain theory that these equations have solutions because we do not know enough about the structure of A. The particular cost structure defined in section 3.6.3, however, is not only strong, but is based on products. Then we know that Alist is well defined and a functor. Elements of this domain are similar to lazy lists,

$$
\begin{array}{rlrl}
\mathsf{lhd}_p^I \circ \mathsf{lnil}_p^I &= \bot_p & \mathsf{lhd}_p^I \circ \mathsf{lcons}_p^I &= \eta_p \circ \pi_1 \\
\mathsf{ltl}_p^I \circ \mathsf{lnil}_p^I &= \bot_p & \mathsf{ltl}_p^I \circ \mathsf{lcons}_p^I &= \pi_2 \\
\mathsf{lhd}_{p'}^I \circ \mathsf{List}^I f &= (\eta \circ f)^* \circ \mathsf{lhd}_p^I & \mathsf{ltl}_{p'}^I \circ \mathsf{List}^I f &= (\eta \circ \mathsf{List}^I f)^* \circ \mathsf{ltl}_p^I \\
\mathsf{lnull?}_p^I \circ \mathsf{lnil}_p^I &= \mathsf{tt} & \mathsf{lnull?}_p^I \circ \mathsf{lcons}_p^I &= \mathsf{ff} \circ ! \\
\mathsf{lnull?}_{p'}^I \circ \mathsf{List}^I f &= \mathsf{lnull?}_p^I & \mathsf{List}^I(h,d) \circ \mathsf{lnil}_p^I &= \mathsf{lnil}_{p'}^I
\end{array}
$$
$$
\mathsf{List}^I(h,d) \circ \mathsf{lcons}_p^I = \mathsf{lcons}_{p'}^I \circ ((h,d) \times (\eta \circ \mathsf{List}^I f)^*)
$$

Figure 4.9: Properties of cost lists

except that each tail that is not $\bot$ contains a cost as well as the actual list tail. The morphism $\mathsf{Alist}\,(f)$, however, is still a simple mapping function applying f to each element in the list.

With these definitions we can define morphisms for the list constants; these are the same as for Llist except that the cost structure A are substituted for the lifting monad L:

$$
\begin{array}{rl}
\mathsf{anil}_X &= \mathsf{afold}_X \circ \mathsf{inl} : \mathbf{1} \to \mathsf{Alist}\,(X) \\
\mathsf{ahd}_X &= [\bot_X, \mathsf{up}_{AX} \circ \eta_X^A \circ \pi_1] \circ \mathsf{aunfold}_X : \mathsf{Alist}\,(X) \to LAX \\
\mathsf{anull?}_X &= [\mathsf{tt}, \mathsf{ff} \circ !] \circ \mathsf{aunfold}_X : \mathsf{Alist}\,(X) \to \mathbf{B} \\
\mathsf{atl}_X &= [\bot_X, \pi_2] \circ \mathsf{aunfold}_X : \mathsf{Alist}\,(X) \to LA\mathsf{Alist}\,(X) \\
\mathsf{acons}_X &= \mathsf{afold}_X \circ \mathsf{inr} : X \times LA\mathsf{Alist}\,(X) \to \mathsf{Alist}\,(X)
\end{array}
$$

The extensional semantics uses the lazy list functor Llist. Therefore we want the extensional part of the arrow category to be Llist. To convert between them, let $f : X \to X'$ be an object in $\mathbf{PDom}^I$. Then let $\gamma_X : \mathsf{Alist}\,(X) \to \mathsf{Llist}\,(X)$ be the (least) solution to the equation.

$$
\gamma_X = \mathsf{lfold}_X \circ (\mathsf{id} + (\mathsf{id} \times L\gamma_X)) \circ (\mathsf{id} + (\mathsf{id} \times L\delta_{\mathrm{Alist}(X)})) \circ \mathsf{aunfold}_X
$$

This morphism removes the cost from each tail within the list. Again, if A is a product functor (with δ a projection) we know that this equation has a solution; it may also have a solution for other arrow cost structures. Furthermore, it can be shown, using the property that limits preserve naturality in $\mathbf{PDom}$, that γ is a natural transformation from Alist to Llist, that is, for any morphism $f : X \to Y$,

$$
\gamma_Y \circ \mathsf{Alist}\,(f) = \mathsf{List}\,(f) \circ \gamma_X
$$

This natural transformation can then be used to define *cost lists* in $\mathbf{PDom}^I$. A cost list is similar to a lazy list except that is built using the cost functors instead of the lifting monad. What this means is not so much that elements of a list have cost, but that costs are paired with each tail. Thus taking the tail of a cost list may add extra cost to the result.

Given an morphism $p : X \to D$ let $\mathsf{List}^I(p) = \mathsf{Llist}\,(p) \circ \gamma_X$ and, given a morphism $(h,d) : p \to p'$, let $\mathsf{List}^I(h,d) = (\mathsf{Alist}\,(h), \mathsf{Llist}\,(d))$. For the list primitives, let $\mathsf{lnil}_p^I = (\mathsf{anil}_X, \mathsf{lnil}_D)$ and so forth. These primitives satisfy the properties listed in Figure 4.9. We will be using only these properties to prove soundness.

To get the meanings of the constants, we take extensional meanings and substitute as we did for the other constants, except that we also substitute the cost list primitives for the lazy list primitives. Each of the non-constructors is associated with a single input-independent cost which we simply add to the equations. The meanings for the list constants is thus as in Figure 4.10.

$$\begin{array}{rcl}
\mathcal{C}^{\mathrm{N}}[\![\mathtt{nil}]\!] & = & \eta \circ \mathsf{lnil}^{I} \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{cons}]\!] & = & \eta \circ \mathsf{lcons}^{I} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{head}]\!] & = & [\![t_{\mathrm{head}}]\!] \circ (\mathsf{id}^{*} \circ \mathsf{lhd}^{I})^{*} \circ \pi_2 \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{tail}]\!] & = & [\![t_{\mathrm{tail}}]\!] \circ \mathsf{ltl}^{I*} \circ \pi_2 \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{nil?}]\!] & = & [\![t_{\mathrm{nil?}}]\!] \circ (\eta \circ \mathsf{lnull?}^{I})^{*} \circ \pi_2
\end{array}$$

Figure 4.10: Call-by-name semantics for list constants in FL

4.4.3 Soundness

The next step needed shows that all of the constants are sound. For the integer and boolean constants the semantics was defined in such a way that soundness of the call-by-name semantics can be derived from the soundness of the call-by-value semantics (and from a straightforward relation between the two operational semantics).

Theorem 4.4.1 *Let c be some constant of arity n such that $c \in \mathbf{Const}_{\tau_1 \to \ldots \to \tau_n \to \tau}$, and the τ_i plus τ are ground types. Also suppose that every call-by-value application rule for c has no premises, and that*

$$\mathrm{vapply}(c, v_1, \ldots v_n) \stackrel{t}{\Rightarrow} v$$

is a call-by-value application rule for c if and only if

$$\frac{e_1 \stackrel{t_1}{\Rightarrow}_{\mathrm{n}} v_1 \ldots e_n \stackrel{t_n}{\Rightarrow}_{\mathrm{n}} v_n}{\mathrm{napply}(c, e_1, \ldots e_n) \stackrel{t+t_n+\ldots+t_1}{\Rightarrow} v}$$

is a call-by-name application rule, and there are no other call-by-name application rules for c with n arguments. Lastly assume that for all values v of ground type g, $\mathcal{N}[\![v : g]\!] = \mathcal{V}[\![v : g]\!]$. Then $f^{*} \circ \psi^{(n)}$ is sound for c in the call-by-name semantics whenever f is sound for c in the call-by-value semantics.

Proof. Let $e_1, \ldots, e_n$ be closed expressions of type $\tau_1, \ldots, \tau_n$ respectively and suppose that

$$\frac{e_1 \stackrel{t_1}{\Rightarrow}_{\mathrm{n}} v_1 \ldots e_n \stackrel{t_n}{\Rightarrow}_{\mathrm{n}} v_n}{\mathrm{napply}(c, e_1, \ldots e_n) \stackrel{t+t_n+\ldots+t_1}{\Rightarrow} v}$$

is an application rule for call-by-name. Then by assumption

$$\mathrm{vapply}(c, v_1, \ldots, v_n) \stackrel{t}{\Rightarrow} v$$

is an application rule for call-by-value. Also we know that for each i, $\mathcal{V}[\![v_i : \tau_i]\!] = \mathcal{N}[\![v_i : \tau_i]\!]$ and $\mathcal{V}[\![v : \tau]\!] = \mathcal{N}[\![v : \tau]\!]$.

Suppose that for each i, $\mathcal{N}[\![e_i : \tau_i]\!] = [\![t_i]\!] \circ \mathcal{N}[\![v_i : \tau_i]\!]$. Then by Lemma 3.4.8, for each i there exists an y_i such that $\mathcal{V}[\![v_i : \tau_i]\!] = \eta \circ y_i$, so by the soundness assumption for call-by-value

$$f \circ \langle\rangle(y_1, \ldots, y_n) = [\![t]\!] \circ \mathcal{V}[\![v : \tau]\!]$$

Therefore

$$\begin{array}{rl} & f^* \circ \psi^{(n)} \circ \langle\rangle(\mathcal{N}[\![e_1 : \tau_1]\!], \ldots, \mathcal{N}[\![e_n : \tau_n]\!]) \\ = & f^* \circ [\![t_n + \ldots + t_1]\!] \circ \eta \circ \langle\rangle(y_1, \ldots, y_n) \\ = & [\![t_n + \ldots + t_1]\!] \circ f \circ \langle\rangle(y_1, \ldots, y_n) \\ = & [\![t_n + \ldots + t_1]\!] \circ [\![t]\!] \circ \mathcal{V}[\![v : \tau]\!] \\ = & [\![t + t_n + \ldots + t_1]\!] \circ \mathcal{N}[\![v : \tau]\!] \end{array}$$

so $f^* \circ \psi^{(n)}$ is sound for c in the call-by-name semantics. □

For the constants of the language FL all the stated assumptions in this theorem hold. All the operational rules for integers in the call-by-value semantics are of the form

$$\text{vapply}(c, v_1, \ldots v_n) \stackrel{t}{\Rightarrow} v$$

and each equivalent rule for the call-by-name semantics is of the form

$$\frac{e_1 \stackrel{t_1}{\Rightarrow}_{\mathrm{n}} v_1 \ldots e_n \stackrel{t_n}{\Rightarrow}_{\mathrm{n}} v_n}{\text{napply}(c, e_1, \ldots e_n) \stackrel{t+t_n+\ldots+t_1}{\Rightarrow} v}$$

Furthermore, the values of ground type in FL are the constants `true`, `false`, and $\overline{n}$ for integers n. By inspection it is clear that their meanings are the same for both call-by-name and call-by-value.

For the other constants each operational rule needs to be examined.

Products

The constant `pair` is sound; it is a strict constructor and its meaning has the form $(\eta \circ f)^* \circ \psi^{(2)}$. For the other constants, it will be useful to know the meaning of $<v_1, v_2>$ for closed values v_1 of type τ_1 and v_2 of type τ_2. As v_1 and v_2 are values, there exist morphisms y_1 and y_2 such that $\mathcal{N}[\![v_1 : \tau_1]\!] = \eta \circ y_1$ and $\mathcal{N}[\![v_2 : \tau_2]\!] = \eta \circ y_2$. Then

$$\begin{array}{rcl} \mathcal{N}[\![<v_1, v_2> : \tau_1 \times \tau_2]\!] & = & \mathcal{C}^{\mathrm{N}}[\![\mathtt{pair}]\!] \circ \langle\rangle(\mathcal{N}[\![v_1 : \tau_1]\!], \mathcal{N}[\![v_2 : \tau_2]\!]) \\ & = & (\eta \circ (\eta \times \eta) \circ (\pi_2 \times \mathsf{id}))^* \circ \psi^{(2)} \circ \langle\rangle(\mathcal{N}[\![v_1 : \tau_1]\!], \mathcal{N}[\![v_2 : \tau_2]\!]) \\ & = & (\eta \circ (\eta \times \eta) \circ (\pi_2 \times \mathsf{id}))^* \circ \eta \circ \langle\rangle(y_1, y_2) \\ & = & \eta \circ (\eta \times \eta) \circ (\pi_2 \times \mathsf{id}) \circ \langle\rangle(y_1, y_2) \\ & = & \eta \circ \langle \eta \circ y_1, \eta \circ y_2 \rangle \\ & = & \eta \circ \langle \mathcal{N}[\![v_1 : \tau_1]\!], \mathcal{N}[\![v_2 : \tau_2]\!] \rangle \end{array}$$

This shows that the meaning of $<v_1, v_2>$ is the pair of the meaning of v_1 and v_2, as expected.

From this equation it is possible to show the soundness of the other product constants from their operational application rules. The proofs for both rules are similar so we only present the proof of the rule for `fst`:

$$\frac{e \stackrel{t}{\Rightarrow}_{\mathrm{n}} <v_1, v_2>}{\text{napply}(\mathtt{fst}, e) \stackrel{t+t_{\mathrm{fst}}}{\Rightarrow} v_1}$$

Because $\mathtt{fst} \in \mathbf{Const}_{(\tau_1 \times \tau_2) \to \tau_1}$, e must be a closed variable of type $\tau_1 \times \tau_2$. Therefore we can assume that $\mathcal{N}[\![e : \tau_1 \times \tau_2]\!] = [\![t]\!] \circ \mathcal{N}[\![<v_1, v_2>]\!] = [\![t]\!] \circ \eta \circ \langle \mathcal{N}[\![v_1 : \tau_1]\!], \mathcal{N}[\![v_2 : \tau_2]\!] \rangle$, so

$$\begin{array}{rl} & \mathcal{C}^{\mathrm{N}}[\![\mathtt{fst}]\!] \circ \langle\rangle(\mathcal{N}[\![e : \tau_1 \times \tau_2]\!]) \\ = & [\![t_{\mathrm{fst}}]\!] \circ {\pi_1}^* \circ \mathcal{N}[\![e : \tau_1 \times \tau_2]\!] \\ = & [\![t_{\mathrm{fst}}]\!] \circ {\pi_1}^* \circ [\![t]\!] \circ \eta \circ \langle \mathcal{N}[\![v_1 : \tau_1]\!], \mathcal{N}[\![v_2 : \tau_2]\!] \rangle \\ = & [\![t_{\mathrm{fst}}]\!] \circ [\![t]\!] \circ \pi_1 \circ \langle \mathcal{N}[\![v_1 : \tau_1]\!], \mathcal{N}[\![v_2 : \tau_2]\!] \rangle \\ = & [\![t + t_{\mathrm{fst}}]\!] \circ \mathcal{N}[\![v_1 : \tau_1]\!] \end{array}$$

Sums

The constants `inl` and `inr` are both sound since they are non-strict constructors that factor through η. For `case` the two relevant application rules need to be examined; however, as the proofs that they are sound are almost identical, we only show the proof for the left-hand case:

$$\frac{e \overset{t}{\Rightarrow}_{\mathrm{n}} \mathtt{inl}(e') \quad e_1(e') \overset{t_1}{\Rightarrow}_{\mathrm{n}} v}{\mathrm{napply}(\mathtt{case}, e, e1, e2) \overset{t_1+t+t_{\mathrm{case}}}{\Rightarrow} v}$$

The type of `case` is $(\tau_1 + \tau_2) \to (\tau_1 \to \tau) \to (\tau_2 \to \tau) \to \tau$ given types τ_1, τ_2, and τ. Therefore e must be a closed expression of type $\tau_1 + \tau_2$, e_1 a closed expression of type $\tau_1 \to \tau$, and e_2 a closed expression of type $\tau_2 \to \tau$. Thus by assumption

$$\begin{aligned}
\mathcal{N}[\![e : \tau_1 + \tau_2]\!] = \;& [\![t]\!] \circ \mathcal{N}[\![\mathtt{inl}(e') : \tau_1 + \tau_2]\!] \\
= \;& [\![t]\!] \circ \mathcal{C}^{\mathrm{N}}[\![\mathtt{inl}]\!] \circ \langle\rangle(\mathcal{N}[\![e' : \tau_1]\!]) \\
= \;& [\![t]\!] \circ \eta \circ \mathsf{inl} \circ \mathcal{N}[\![e' : \tau_1]\!]
\end{aligned}$$

and $\mathcal{N}[\![e_1(e') : \tau]\!] = [\![t_1]\!] \circ \mathcal{N}[\![v : \tau]\!]$. Next note that for any morphisms f_1, f_2, y, y' and any cost t,

$$\begin{aligned}
&(\mathsf{case}(f_1, f_2))^* \circ \psi \circ \langle [\![t]\!] \circ \eta \circ \mathsf{inl} \circ y, \eta \circ y' \rangle \\
&\quad = \; (\mathsf{case}(f_1, f_2))^* \circ [\![t]\!] \circ \eta \circ \langle \mathsf{inl} \circ y, y' \rangle \\
&\quad = \; [\![t]\!] \circ \mathsf{case}(f_1, f_2) \circ (\mathsf{inl} \times \mathsf{id}) \circ \langle y, y' \rangle \\
&\quad = \; [\![t]\!] \circ f_1 \circ \langle y, y' \rangle
\end{aligned}$$

and for any morphisms f, f_1, f_2,

$$\begin{aligned}
\mathsf{nleft}^{\mathrm{I}} \circ \langle f, \langle f_1, f_2 \rangle \rangle = \;& \mathsf{app}^* \circ \psi \circ \beta \circ (\eta \times \pi_1) \circ \langle f, \langle f_1, f_2 \rangle \rangle \\
= \;& \mathsf{app}^* \circ \psi \circ \langle f_1, \eta \circ f \rangle
\end{aligned}$$

Thus

$$\begin{aligned}
&\mathsf{nleft}^{\mathrm{I}} \circ \langle \mathcal{N}[\![e' : \tau_1]\!], \langle \mathcal{N}[\![e_1 : \tau_1 \to \tau]\!], \mathcal{N}[\![e_2 : \tau_1 \to \tau]\!] \rangle \rangle \\
&\quad = \; \mathsf{app}^* \circ \psi \circ \langle \mathcal{N}[\![e_1 : \tau_1 \to \tau]\!], \eta \circ \mathcal{N}[\![e' : \tau_1]\!] \rangle \\
&\quad = \; \mathcal{N}[\![e_1(e') : \tau]\!] \\
&\quad = \; [\![t_1]\!] \circ \mathcal{N}[\![v : \tau]\!]
\end{aligned}$$

so

$$\begin{aligned}
&\mathcal{C}^{\mathrm{N}}[\![\mathsf{case}]\!] \circ \langle\rangle(\mathcal{N}[\![e : \tau_1 + \tau_2]\!], \mathcal{N}[\![e_1 : \tau_1 \to \tau]\!], \mathcal{N}[\![e_2 : \tau_1 \to \tau]\!]) \\
&\quad = \; [\![t_{\mathrm{case}}]\!] \circ (\mathsf{case}(\mathsf{nleft}^{\mathrm{I}}, \mathsf{nright}^{\mathrm{I}}))^* \circ \psi \circ (\pi_2 \times \eta) \circ \alpha^r \\
&\qquad\quad \circ \langle\rangle(\mathcal{N}[\![e : \tau_1 + \tau_2]\!], \mathcal{N}[\![e_1 : \tau_1 \to \tau]\!], \mathcal{N}[\![e_2 : \tau_1 \to \tau]\!]) \\
&\quad = \; [\![t_{\mathrm{case}}]\!] \circ (\mathsf{case}(\mathsf{nleft}^{\mathrm{I}}, \mathsf{nright}^{\mathrm{I}}))^* \circ \psi \circ \langle \mathcal{N}[\![e : \tau_1 + \tau_2]\!], \\
&\qquad\quad \eta \circ \langle \mathcal{N}[\![e_1 : \tau_1 \to \tau]\!], \mathcal{N}[\![e_2 : \tau_1 \to \tau]\!] \rangle \rangle \\
&\quad = \; [\![t_{\mathrm{case}}]\!] \circ (\mathsf{case}(\mathsf{nleft}^{\mathrm{I}}, \mathsf{nright}^{\mathrm{I}}))^* \circ \psi \circ \langle [\![t]\!] \circ \eta \circ \mathsf{inl} \circ \mathcal{N}[\![e' : \tau_1]\!], \\
&\qquad\quad \eta \circ \langle \mathcal{N}[\![e_1 : \tau_1 \to \tau]\!], \mathcal{N}[\![e_2 : \tau_1 \to \tau]\!] \rangle \rangle \\
&\quad = \; [\![t_{\mathrm{case}}]\!] \circ [\![t]\!] \circ \mathsf{nleft}^{\mathrm{I}} \circ \langle \mathcal{N}[\![e' : \tau_1]\!], \langle \mathcal{N}[\![e_1 : \tau_1 \to \tau]\!], \mathcal{N}[\![e_2 : \tau_1 \to \tau]\!] \rangle \rangle \\
&\quad = \; [\![t + t_{\mathrm{case}}]\!] \circ [\![t_1]\!] \circ \mathcal{N}[\![v : \tau]\!] \\
&\quad = \; [\![t_1 + t + t_{\mathrm{case}}]\!] \circ \mathcal{N}[\![v : \tau]\!]
\end{aligned}$$

Thus `case` is sound.

Lists

The constants `cons` and `nil` are sound because they are non-strict constructors that factor through η. The other constants need to be compared with the operational semantics: two rules for `nil?` and one rule each for `head` and `tail` (as there are no rules listed for `head` and `tail` on `nil`).

To prove soundness for non-constructor constants it is be helpful to know the meaning of $e_1{::}e_2$. Let e_1 be a closed expression of type τ and e_2 be a closed expression of type $\mathbf{list}(\tau)$. Then

$$\begin{aligned}\mathcal{N}[\![e_1{::}e_2 : \mathbf{list}(\tau)]\!] &= \mathcal{C}^{\mathrm{N}}[\![\mathsf{cons}]\!] \circ \langle\rangle(\mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]) \\ &= \eta \circ \mathsf{lcons}^I \circ (\pi_2 \times \mathsf{id}) \circ \langle\rangle(\mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]) \\ &= \eta \circ \mathsf{lcons}^I \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle\end{aligned}$$

We then examine the rules individually:

Case $\dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} e_1{::}e_2 \qquad e_1 \overset{t_1}{\Rightarrow}_{\mathrm{n}} v}{\mathrm{napply}(\mathtt{head}, e) \overset{t_1+t+t_{\mathrm{head}}}{\Rightarrow} v}$**:**

For some type τ, e and e_2 must be closed expressions of type $\mathbf{list}(\tau)$ and e_1 must be a closed expression of type τ. Then by assumption

$$\begin{aligned}\mathcal{N}[\![e : \mathbf{list}(\tau)]\!] &= [\![t]\!] \circ \mathcal{N}[\![e_1{::}e_2 : \mathbf{list}(\tau)]\!] \\ &= [\![t]\!] \circ \eta \circ \mathsf{lcons}^I \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle\end{aligned}$$

and $\mathcal{N}[\![e_1 : \tau]\!] = [\![t_1]\!] \circ \mathcal{N}[\![v : \tau]\!]$. Next, note that

$$\begin{aligned}(\mathsf{id}^* \circ \mathsf{lhd}^I)^* \circ [\![t]\!] \circ \eta \circ \mathsf{lcons}^I &= [\![t]\!] \circ \mathsf{id}^* \circ \mathsf{lhd}^I \circ \mathsf{lcons}^I \\ &= [\![t]\!] \circ \mathsf{id}^* \circ \eta \circ \pi_1 \\ &= [\![t]\!] \circ \pi_1\end{aligned}$$

Therefore

$$\begin{aligned}&\mathcal{C}^{\mathrm{N}}[\![\mathtt{head}]\!] \circ \langle\rangle(\mathcal{N}[\![e : \mathbf{list}(\tau)]\!]) \\ &\quad= [\![t_{\mathrm{head}}]\!] \circ (\mathsf{id}^* \circ \mathsf{lhd}^I)^* \circ \pi_2 \circ \langle\rangle(\mathcal{N}[\![e : \mathbf{list}(\tau)]\!]) \\ &\quad= [\![t_{\mathrm{head}}]\!] \circ (\mathsf{id}^* \circ \mathsf{lhd}^I)^* \circ [\![t]\!] \circ \eta \circ \mathsf{lcons}^I \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle \\ &\quad= [\![t_{\mathrm{head}}]\!] \circ [\![t]\!] \circ \pi_1 \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle \\ &\quad= [\![t + t_{\mathrm{head}}]\!] \circ \mathcal{N}[\![e_1 : \tau]\!] \\ &\quad= [\![t_1 + t + t_{\mathrm{head}}]\!] \circ \mathcal{N}[\![v : \tau]\!]\end{aligned}$$

so `head` is sound.

Case $\dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} e_1{::}e_2 \qquad e_2 \overset{t_2}{\Rightarrow}_{\mathrm{n}} v}{\mathrm{napply}(\mathtt{tail}, e) \overset{t_2+t+t_{\mathrm{tail}}}{\Rightarrow} v}$**:**

Again for some type τ, e and e_2 have type $\mathbf{list}(\tau)$ and e_1 has type τ. Therefore by assumption

$$\mathcal{N}[\![e : \mathbf{list}(\tau)]\!] = [\![t]\!] \circ \eta \circ \mathsf{lcons}^I \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle$$

as before, and $\mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!] = [\![t_2]\!] \circ \mathcal{N}[\![v : \mathbf{list}(\tau)]\!]$ so

$$\begin{aligned}
&\mathcal{C}^{\mathrm{N}}[\![\texttt{tail}]\!] \circ \langle\rangle(\mathcal{N}[\![e : \mathbf{list}(\tau)]\!]) \\
&\quad= [\![t_{\mathrm{head}}]\!] \circ (\mathsf{ltl}^I)^* \circ [\![t]\!] \circ \eta \circ \mathsf{lcons}^I \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle \\
&\quad= [\![t_{\mathrm{head}}]\!] \circ [\![t]\!] \circ \mathsf{ltl}^I \circ \mathsf{lcons}^I \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle \\
&\quad= [\![t + t_{\mathrm{head}}]\!] \circ \pi_2 \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle \\
&\quad= [\![t_2 + t + t_{\mathrm{head}}]\!] \circ \mathcal{N}[\![v : \tau]\!]
\end{aligned}$$

Thus `tail` is sound.

Case $\dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} \texttt{nil}}{\mathrm{napply}(\texttt{nil?}, e) \overset{t+t_{\mathrm{nil?}}}{\Rightarrow} \texttt{true}}$**:**

For some type τ, e must be a closed expression of type $\mathbf{list}(\tau)$. Therefore by assumption

$$\mathcal{N}[\![e : \mathbf{list}(\tau)]\!] = [\![t]\!] \circ \mathcal{N}[\![\texttt{nil} : \mathbf{list}(\tau)]\!] = [\![t]\!] \circ \eta \circ \mathsf{lnil}^I$$

so

$$\begin{aligned}
&\mathcal{N}[\![\texttt{nil?}]\!] \circ \langle\rangle(\mathcal{N}[\![e : \mathbf{list}(\tau)]\!]) \\
&\quad= [\![t_{\mathrm{nil?}}]\!] \circ (\eta \circ \mathsf{lnull?}^I)^* \circ \pi_2 \circ \langle\rangle([\![t]\!] \circ \eta \circ \mathsf{lnil}^I) \\
&\quad= [\![t_{\mathrm{nil?}}]\!] \circ (\eta \circ \mathsf{lnull?}^I)^* \circ [\![t]\!] \circ \eta \circ \mathsf{lnil}^I \\
&\quad= [\![t_{\mathrm{nil?}}]\!] \circ [\![t]\!] \circ \eta \circ \mathsf{lnull?}^I \circ \mathsf{lnil}^I \\
&\quad= [\![t + t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathsf{tt} \\
&\quad= [\![t + t_{\mathrm{nil?}}]\!] \circ \mathcal{N}[\![\texttt{true} : \mathbf{bool}]\!]
\end{aligned}$$

Case $\dfrac{e \overset{t}{\Rightarrow}_{\mathrm{n}} e_1{::}e_2}{\mathrm{napply}(\texttt{nil?}, e) \overset{t+t_{\mathrm{nil?}}}{\Rightarrow} \texttt{false}}$**:**

Again for some τ, e and e_2 must be closed expressions of type $\mathbf{list}(\tau)$ and e_1 must be a closed expression of type $\mathbf{list}(\tau)$. Then by assumption

$$\mathcal{N}[\![e : \mathbf{list}(\tau)]\!] = [\![t]\!] \circ \mathcal{N}[\![e_1{::}e_2]\!] = [\![t]\!] \circ \eta \circ \mathsf{lcons}^I \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle$$

so

$$\begin{aligned}
&\mathcal{N}[\![\texttt{nil?}]\!] \circ \langle\rangle(\mathcal{N}[\![e : \mathbf{list}(\tau)]\!]) \\
&\quad= [\![t_{\mathrm{nil?}}]\!] \circ (\eta \circ \mathsf{lnull?}^I)^* \circ \mathcal{N}[\![e : \mathbf{list}(\tau)]\!] \\
&\quad= [\![t_{\mathrm{nil?}}]\!] \circ (\eta \circ \mathsf{lnull?}^I)^* \circ [\![t]\!] \circ \eta \circ \mathsf{lcons}^I \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle \\
&\quad= [\![t_{\mathrm{nil?}}]\!] \circ [\![t]\!] \circ \eta \circ \mathsf{lnull?}^I \circ \mathsf{lcons}^I \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle \\
&\quad= [\![t + t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathsf{ff} \circ ! \circ \langle \mathcal{N}[\![e_1 : \tau]\!], \mathcal{N}[\![e_2 : \mathbf{list}(\tau)]\!]\rangle \\
&\quad= [\![t + t_{\mathrm{nil?}}]\!] \circ \mathcal{N}[\![\texttt{false}]\!]
\end{aligned}$$

Thus `nil?` is sound.

4.5 Example programs

In this section we examine a few of the small programs used as examples in Chapter 3. With these programs we intend to show not only the difference between the two evaluation strategies but also the problems that arise when call-by-name is used without the ability to re-use the result of an evaluation. We also examine a few other programs that use infinite lists.

There are abbreviations that can, again, significantly simplify the evaluation of examples. For example, the meaning of a conditional expression is the same in the call-by-name semantics as in the call-by-value semantics. Therefore we can use the same abbreviations:

$$\begin{aligned}
\mathbf{bool}(x) &= \begin{cases} \mathsf{tt} & \text{if } x \text{ is true} \\ \mathsf{ff} & \text{otherwise} \end{cases} \\
\mathbf{cond}(f, g, h) &= \mathsf{cond}^* \circ \psi \circ \langle f, \eta \circ \langle g, h \rangle \rangle \\
\mathbf{cond}(\{x\}, g, h) &= \mathbf{cond}(\eta \circ \mathbf{bool}(x), g, h)
\end{aligned}$$

When abbreviating applications, the meaning differs, but we can use still use the function **apply**, where $\mathbf{apply}(f) = f$ and

$$\mathbf{apply}(f, f_1, \ldots, f_n) = \mathsf{app}^* \circ \psi \circ \langle \mathbf{apply}(f, f_1, \ldots, f_{n-1}, \eta \circ f_n) \rangle$$

The difference is that when we used $\mathbf{apply}(f, f_1, \ldots, f_n)$ for the call-by-value, we assumed that each f_i was a value, i.e., a morphism to $\mathcal{T}^{\mathrm{V}}[\![\tau]\!]$ for some type τ. For the call by name we instead assume that each f_i can contain cost, i.e., it is a morphism to $C\mathcal{T}^{\mathrm{N}}[\![\tau]\!]$.

Because the definition of **apply** is similar to the definition of application for the call-by-name semantics, it is clear that

$$\mathcal{N}[\![\Gamma \vdash ee_1 \ldots e_n : \tau]\!] = \mathbf{apply}(\mathcal{N}[\![\Gamma \vdash e : \tau_1 \to \ldots \to \tau_n \to \tau]\!], \mathcal{N}[\![\Gamma \vdash e_1 : \tau_1]\!], \ldots, \mathcal{N}[\![\Gamma \vdash e_n : \tau_n]\!])$$

Similarly, if the arity of c is at least n,

$$\mathbf{apply}(\mathcal{N}[\![c]\!], f_1, \ldots, f_n) = \mathcal{C}^{\mathrm{N}}[\![c]\!] \circ \langle\rangle(f_1, \ldots, f_n)$$

We also have a similar result when applying abstractions:

Theorem 4.5.1 *Suppose that* $f = [\![t]\!] \circ \mathcal{N}[\![\Gamma \vdash \texttt{lam}\ x_1.\ldots\texttt{lam}\ x_n.e : \tau_0]\!] \circ \langle\rangle(h_1, \ldots, h_k)$*, where* τ_0 *equals* $\tau_1 \to \ldots \to \tau_n \to \tau$*, and* $\langle\rangle(h_1, \ldots, h_n) : X \to \mathcal{T}^{\mathrm{N}}[\![\Gamma]\!]$*. Furthermore suppose that for* $1 \le i \le n$*,* $f_i : X \to C\mathcal{T}^{\mathrm{N}}[\![\tau_i]\!]$*. Then*

$$\begin{aligned}
&\mathbf{apply}(f, f_1, \ldots, f_i) \\
&\quad = [\![i\ t_{\mathrm{app}} + t]\!] \circ \mathcal{N}[\![\Gamma, x_1 : \tau_1, \ldots, x_i : \tau_i \vdash \texttt{lam}\ x_{i+1}.\ldots\texttt{lam}\ x_n.e : \tau_{i+1} \to \ldots \to \tau_n \to \tau]\!] \\
&\qquad \circ \langle\rangle(h_1, \ldots, h_k, f_1, \ldots, f_n)
\end{aligned}$$

Proof. By straightforward induction on i. □

Unlike in the call-by-value version of this theorem, we do not need to assume that + is commutative for the result to hold. We still, however, assume commutativity when calculating the cost of the examples.

$$\begin{array}{rcl}
\mathcal{C}^{\mathrm{N}}[\![\mathtt{head}]\!] \circ \langle\rangle([\![t]\!] \circ \eta \circ [\,]) & = & \bot \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{head}]\!] \circ \langle\rangle([\![t]\!] \circ \eta \circ [\langle z, r\rangle +]) & = & [\![t + t_{\mathrm{head}}]\!] \circ z \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{tail}]\!] \circ \langle\rangle([\![t]\!] \circ \eta \circ [\,]) & = & \bot \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{tail}]\!] \circ \langle\rangle([\![t]\!] \circ \eta \circ [\langle z, \bot\rangle]) & = & \bot \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{tail}]\!] \circ \langle\rangle([\![t]\!] \circ \eta \circ [\langle z_1, t_1\rangle, \langle z_2, r\rangle +]) & = & [\![t_1 + t + t_{\mathrm{tail}}]\!] \circ \eta \circ [\langle z_2, r\rangle +] \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{nil?}]\!] \circ \langle\rangle([\![t]\!] \circ \eta \circ [\,]) & = & [\![t + t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathsf{tt} \\
\mathcal{C}^{\mathrm{N}}[\![\mathtt{nil?}]\!] \circ \langle\rangle([\![t]\!] \circ \eta \circ [\langle z, r\rangle +]) & = & [\![t + t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathsf{ff}
\end{array}$$

Figure 4.11: FL list properties

4.5.1 Lists

The abbreviations used for lists in the previous chapter are insufficient for cost lists. In the first place, some lists may be infinite. Secondly, and more importantly, there may be extra costs involved in evaluating the tail of the list and we need to be able to represent that cost.

Therefore let $[\,] = \mathsf{lnil}^I$, $[\langle z, \bot\rangle] = \mathsf{lcons}^I \circ \langle z, \bot\rangle$, and $[\langle z, t\rangle] = \mathsf{lcons}^I \circ \langle z, [\![t]\!] \circ \eta \circ \mathsf{lnil}^I\rangle$. We will use r (as in $[\langle z, r\rangle]$) to represent either $\bot$ or t for some $t \in T$. Next let

$$[\langle z_1, t_1\rangle, \ldots, \langle z_{n-1}, t_{n-1}\rangle, \langle z_n, r\rangle] = \mathsf{lcons}^I \circ \langle z_1, [\![t_1]\!] \circ \eta \circ [\langle z_2, t_2\rangle, \ldots, \langle z_n, r\rangle]\rangle$$

Note that each z_i represents the i'th element of the list (which itself may be contain cost), while each t_i represent the extra cost of taking the i'th tail. For infinite lists we write

$$[\langle z_1, t_1\rangle, \langle z_2, t_2\rangle, \ldots] = \bigsqcup_{n=1}^{\infty} [\langle z_1, t_1\rangle, \ldots, \langle z_n, \bot\rangle]$$

For a list that has at least n elements and may be infinite or finite we write $[\langle z_1, t_1\rangle, \ldots, \langle z_n, r\rangle +]$.

It can be shown that the meaning of all closed expressions of type $\mathbf{list}(\tau)$ can be represented by one of the above forms. For example, the meaning of $\mathtt{rec}\ \mathtt{z}.1{::}\mathtt{z}$ (the infinite list of 1's) is

$$[\![t_{\mathrm{rec}}]\!] \circ \eta \circ [\langle \eta \circ \mathsf{n}_1{}^I, t_{\mathrm{rec}}\rangle, \langle \eta \circ \mathsf{n}_1{}^I, t_{\mathrm{rec}}\rangle, \ldots]$$

Note that the meaning shows not only that the list is an infinite list of 1's, but that each element of the list is a value and there is a cost of t_{rec} (that is, unrolling the recursion) for evaluating each tail.

Figure 4.11 lists properties of the FL constants and the cost list abbreviations.

4.5.2 The `twice` program revisited

First we look at the simple higher order function

$$\mathtt{twice} = \mathtt{lam}\ \mathtt{f}.\mathtt{lam}\ \mathtt{x}.\mathtt{f}(\mathtt{f}(\mathtt{x}))$$

Let $z_f : \mathbf{1} \to C\mathcal{T}^{\mathrm{N}}[\![\tau \to \tau]\!]$ and $z_x : \mathbf{1} \to C\mathcal{T}^{\mathrm{N}}[\![\tau]\!]$. Then

$$\begin{array}{rcl}
\mathbf{apply}(\mathcal{N}[\![\mathtt{twice}]\!], z_f, z_x) & = & [\![2t_{\mathrm{app}}]\!] \circ \mathcal{N}[\![\mathtt{f} : \tau \to \tau, \mathtt{x} : \tau \vdash \mathtt{f}(\mathtt{f}(\mathtt{x})) : \tau]\!] \circ \langle\rangle(z_f, z_x) \\
& = & [\![2t_{\mathrm{app}}]\!] \circ \mathbf{apply}(z_f, \mathbf{apply}(z_f, z_x))
\end{array}$$

Now suppose z_f refers to a program that does not evaluate its argument; that is, there exists a cost t_c and a morphism $y : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}[\![\tau]\!]$ such that for any z_x, $\mathbf{apply}(z_f, z_x) = [\![t_c]\!] \circ \eta \circ y$. Then

$$\mathbf{apply}(\mathcal{N}[\![\mathtt{twice}]\!], z_f, z_x) = [\![2t_{\mathrm{app}} + t_c]\!] \circ \eta \circ y$$

which shows that the inner `f` is also never applied.

Next suppose that z_f refers to a program such that there exists a cost t_c and a function F_v on values such that for any $z_x = [\![t_x]\!] \circ \eta \circ y_x$, $\mathbf{apply}(z_f, z_x) = [\![t_c + nt_x]\!] \circ \eta \circ F_v(y_x)$. This is a function that evaluates its argument n times, and has an additional constant cost of t_c. For example, z_f could be $\mathcal{N}[\![\mathtt{lam}\ x.x + \ldots + x]\!]$. In that case

$$\begin{aligned} \mathbf{apply}(\mathcal{N}[\![\mathtt{twice}]\!], z_f, z_x) &= [\![2t_{\mathrm{app}}]\!] \circ \mathbf{apply}(z_f, [\![t_c + nt_x]\!] \circ \eta \circ F_v(g_x)) \\ &= [\![2t_{\mathrm{app}} + t_c + n(t_c + nt_x)]\!] \circ \eta \circ F_v(F_v(g_x)) \\ &= [\![2t_{\mathrm{app}} + (n+1)t_c + n^2 t_x]\!] \circ \eta \circ F_v(F_v(g_x)) \end{aligned}$$

In this case the number of times z_x is evaluated increases rapidly with n, as the number of times z_f is applied is $n + 1$.

4.5.3 The `length` program revisited

Again we look at the program

$$\mathtt{length} = \mathtt{rec\ len.lam\ l.if\ nil?(l)\ then}\ \overline{0}\ \mathtt{else}\ \overline{1} + \mathtt{len(tail(l))}$$

which finds the length of a list. Let $\tau_0 = \mathbf{list}(\tau) \to \mathbf{nat}$, and

$$e = \mathtt{if\ nil?(l)\ then}\ \overline{0}\ \mathtt{else}\ \overline{1} + \mathtt{len(tail(l))}$$

Then $\mathtt{length} = \mathtt{rec\ len.lam\ l}.e$, and

$$\mathcal{N}[\![\mathtt{length}]\!] = [\![t_{\mathrm{rec}}]\!] \circ \mathcal{N}[\![\mathtt{len} : \tau_0 \vdash \mathtt{lam\ l}.e : \tau_0]\!] \circ \langle\rangle(\mathcal{N}[\![\mathtt{length}]\!])$$

Let $\Gamma = \mathtt{len} : \tau_0, \mathtt{l} : \mathbf{list}(\tau)$. Then, for any $z : \mathbf{1} \to C\mathcal{T}^{\mathrm{N}}[\![\mathbf{list}(\tau)]\!]$,

$$\mathbf{apply}(\mathcal{N}[\![\mathtt{length}]\!], z) = [\![t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathcal{N}[\![\Gamma \vdash e : \mathbf{nat}]\!] \circ \langle\rangle(\mathcal{N}[\![\mathtt{length}]\!], z)$$

If $z = \perp$, then

$$\mathcal{N}[\![\Gamma \vdash \mathtt{nil?(l)} : \mathbf{bool}]\!] \circ \langle\rangle(\mathcal{N}[\![\mathtt{length}]\!], z) = \mathcal{C}^{\mathrm{N}}[\![\mathtt{nil?}]\!] \circ \langle\rangle(\perp) = \perp$$

so $\mathbf{apply}(\mathcal{N}[\![\mathtt{length}]\!], z) = \perp$ as well. For $z = [\![t]\!] \circ \eta \circ \mathsf{lnil}^I$,

$$\mathcal{N}[\![\Gamma \vdash \mathtt{nil?(l)} : \mathbf{bool}]\!] \circ \langle\rangle(\mathcal{N}[\![\mathtt{length}]\!], z) = [\![t + t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathbf{tt}$$

so

$$\begin{aligned} &\mathbf{apply}(\mathcal{N}[\![\mathtt{length}]\!], z) \\ &\quad= [\![t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ\ \mathbf{cond}(\mathcal{N}[\![\Gamma \vdash \mathtt{nil?(l)} : \mathbf{bool}]\!] \circ \langle\rangle(\mathcal{N}[\![\mathtt{length}]\!], z))\ \mathbf{of} \\ &\qquad\qquad \mathbf{True:}\quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{n_0}^I \circ \langle\rangle(\mathcal{N}[\![\mathtt{length}]\!], z) \\ &\qquad\qquad \mathbf{False:}\quad [\![t_{\mathrm{false}}]\!] \circ \mathcal{N}[\![\Gamma \vdash \overline{1} + \mathtt{len(tail(l))} : \mathbf{nat}]\!] \circ \langle\rangle(\mathcal{N}[\![\mathtt{length}]\!], z) \\ &\quad= [\![t + t_{\mathrm{nil?}} + t_{\mathrm{true}} + t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{n_0}^I \end{aligned}$$

For $z = [\![t]\!] \circ \eta \circ \mathsf{lcons}^I \circ \langle a, z' \rangle$,

$$\mathcal{N}[\![\texttt{nil?}]\!] \circ \langle\rangle(z) = [\![t + t_{\text{nil?}}]\!] \circ \eta \circ \mathsf{ff}$$

and

$$\mathcal{N}[\![\texttt{tail}]\!] \circ \langle\rangle(z) = [\![t + t_{\text{tail}}]\!] \circ z'$$

Therefore

$$\begin{array}{rll} \multicolumn{3}{l}{\mathbf{apply}(\mathcal{N}[\![\texttt{length}]\!], z)} \\ = & [\![t_{\text{app}} + t_{\text{rec}}]\!] \circ & \mathbf{cond}([\![t + t_{\text{nil?}}]\!] \circ \eta \circ \mathsf{ff})\ \mathbf{of} \\ & & \quad \mathbf{True:} \quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{n_0}^I \\ & & \quad \mathbf{False:} \quad [\![t_{\text{false}}]\!] \circ \mathcal{C}^{\mathrm{N}}[\![+]\!] \circ \langle\rangle(\eta \circ \mathsf{n_1}^I, \mathbf{apply}(\mathcal{N}[\![\texttt{length}]\!], [\![t + t_{\text{tail}}]\!] \circ z')) \\ = & \multicolumn{2}{l}{[\![t_{\text{false}} + t + t_{\text{nil?}} + t_{\text{app}} + t_{\text{rec}}]\!] \circ \mathcal{C}^{\mathrm{N}}[\![+]\!] \circ \langle\rangle(\eta \circ \mathsf{n_1}^I, \mathbf{apply}(\mathcal{N}[\![\texttt{length}]\!], [\![t + t_{\text{tail}}]\!] \circ z'))} \end{array}$$

Again if $z' = \perp$ then $\mathbf{apply}(\mathcal{N}[\![\texttt{length}]\!], [\![t + t_{\text{tail}}]\!] \circ z') = \perp$, so $\mathbf{apply}(\mathcal{N}[\![\texttt{length}]\!], z) = \perp$ as well. Therefore for all lists of the form $l = [\langle z_1, t_1 \rangle, \ldots, \langle z_n, \perp \rangle]$, $\mathbf{apply}(\mathcal{N}[\![\texttt{length}]\!], [\![t]\!] \circ \eta \circ l) = \perp$. By continuity `length` applied to infinite lists is also $\perp$.

Therefore we are only interested in lists of the form $[\langle z_1, t_1 \rangle, \ldots, \langle z_n, t_n \rangle]$. In that case, for $n > 0$,

$$\begin{array}{rl} \multicolumn{2}{l}{\mathbf{apply}(\mathcal{N}[\![\texttt{length}]\!], [\![t]\!] \circ \eta \circ [\langle z_1, t_1 \rangle, \ldots, \langle z_n, t_n \rangle])} \\ = & [\![t_{\text{false}} + t + t_{\text{nil?}} + t_{\text{app}} + t_{\text{rec}}]\!] \circ \mathcal{C}^{\mathrm{N}}[\![+]\!] \\ & \circ \langle\rangle(\eta \circ \mathsf{n_1}^I, \mathbf{apply}(\mathcal{N}[\![\texttt{length}]\!], [\![t_1 + t + t_{\text{tail}}]\!] \circ \eta \circ [\langle z_2, t_2 \rangle, \ldots, \langle z_n, t_n \rangle])) \end{array}$$

To find the cost function, suppose that there exist a function $N(i)$ and, for each i, functions $T_i(t', t'_1, \ldots, t'_i)$ such that for any t',

$$\mathbf{apply}(\mathcal{N}[\![\texttt{length}]\!], [\![t']\!] \circ \eta \circ [\langle z_2, t_2 \rangle, \ldots, \langle z_n, t_n \rangle]) = [\![T_{n-1}(t', t_2, \ldots, t_n)]\!] \circ \eta \circ \mathsf{n}_{N(n-1)}{}^I$$

For the case where $n = 1$, $T_0(t') = t' + t_{\text{nil?}} + t_{\text{true}} + t_{\text{app}} + t_{\text{rec}}$ and $N(0) = 0$. Then

$$\begin{array}{rl} \multicolumn{2}{l}{\mathbf{apply}(\mathcal{N}[\![\texttt{length}]\!], [\![t]\!] \circ \eta \circ [\langle z_1, t_1 \rangle, \ldots, \langle z_n, t_n \rangle])} \\ = & [\![t_{\text{false}} + t + t_{\text{nil?}} + t_{\text{app}} + t_{\text{rec}}]\!] \circ \mathcal{C}^{\mathrm{N}}[\![+]\!] \\ & \quad \circ \langle\rangle(\eta \circ \mathsf{n_1}^I, [\![T_{n-1}(t_1 + t + t_{\text{tail}}, t_2, \ldots, t_n)]\!] \circ \eta \circ \mathsf{n}_{N(n-1)}{}^I) \\ = & [\![t_{\text{false}} + t + t_{\text{nil?}} + t_{\text{app}} + t_{\text{rec}} + t_+ + T_{n-1}(t_1 + t + t_{\text{tail}}, t_2, \ldots, t_n)]\!] \\ & \quad \circ \eta \circ \mathsf{n}_{N(n-1)+1}{}^I \end{array}$$

Therefore the cost of `length` is the solution to the equations

$$\begin{array}{rcl} T_0(t) & = & t + t_{\text{nil?}} + t_{\text{true}} + t_{\text{app}} + t_{\text{rec}} \\ T_n(t, t_1, \ldots, t_n) & = & t + t_{\text{nil?}} + t_{\text{false}} + t_{\text{app}} + t_{\text{rec}} + t_+ + T_{n-1}(t_1 + t + t_{\text{tail}}, t_2, \ldots, t_n) \end{array}$$

where $t_1, \ldots, t_n$ are the additional costs within the list. The solutions to these equations,

$$T_n(t, t_1, \ldots, t_n) = (n+1)(t + t_{\text{nil?}} + t_{\text{app}} + t_{\text{rec}} + t_+) + nt_{\text{false}} + \frac{n(n+1)}{2} t_{\text{tail}} + t_{\text{true}} + \sum_{i=1}^{n} (n - i + 1) t_i$$

is $\mathcal{O}(n^2)$. This is substantially less efficient than the call-by-value version of `length`, which was $\mathcal{O}(n)$. The extra cost comes from the repeated calculations of the list and its tails. The first time `length` is called, the list is evaluated. In the recursive calls, not only is the list evaluated, but all the tails of the list must be evaluated each time as well. Therefore we end up with $\mathcal{O}(n^2)$ calls to `tail` instead of $\mathcal{O}(n)$.

To solve this problem we need to add an ability to force the evaluation of an expression and allow the result to be re-used. This is usually done by adding a `let` construct, $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, with the following operational semantics:

$$\frac{e_1 \Rightarrow_{\mathrm{n}} v_1 \quad [v_1/x]e_2 \Rightarrow_{\mathrm{n}} v}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow_{\mathrm{n}} v}$$

Rather than add a new construct to our language directly, we add a new constant `vapp` with the type $(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$, which simulates call-by-value application:

$$\frac{e_2 \overset{t_1}{\Rightarrow}_{\mathrm{n}} v_2 \qquad e_1(v_2) \overset{t_2}{\Rightarrow}_{\mathrm{n}} v}{\mathrm{napply}(\texttt{vapp}, e_1, e_2) \overset{t_2+t_1}{\Rightarrow} v}$$

We can then define $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ to mean $\texttt{vapp}\ (\texttt{lam}\ x.e_2)\ e_1$.

For the denotational semantics, let $\mathcal{C}^{\mathrm{N}}[\![\texttt{vapp}]\!] : \times(C\mathcal{T}^{\mathrm{N}}[\![\tau_1 \to \tau_2]\!], C\mathcal{T}^{\mathrm{N}}[\![\tau_1]\!]) \to C\mathcal{T}^{\mathrm{N}}[\![\tau_2]\!]$ be defined as follows:

$$\mathcal{C}^{\mathrm{N}}[\![\texttt{vapp}]\!] = (\mathbf{apply}(\pi_1, \eta \circ \pi_2))^* \circ \psi \circ (\eta \times \mathsf{id}) \circ (\pi_2 \times \mathsf{id})$$

There are simpler definitions possible for `vapp`, but we cannot easily prove them sound for all possible values of $\mathcal{N}[\![e_1]\!]$. To see that this definition is sound, suppose that $\texttt{vapp}\ e_1\ e_2$ has type τ. Then there exists a type τ' such that $\vdash e_1 : \tau' \to \tau$ and $\vdash e_2 : \tau'$. Next suppose that $\mathcal{N}[\![e_2 : \tau']\!] = [\![t_1]\!] \circ \mathcal{N}[\![v_2 : \tau']\!]$ and $\mathcal{N}[\![e_1(v_2) : \tau]\!] = [\![t_2]\!] \circ \mathcal{N}[\![v : \tau]\!]$. Because v_2 is a value we know that there exists a morphism $y : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}[\![\tau']\!]$ such that $\mathcal{N}[\![v_2 : \tau']\!] = \eta \circ y$. Therefore

$$\begin{aligned}
&\psi \circ (\eta \times \mathsf{id}) \circ (\pi_2 \times \mathsf{id}) \circ \langle\rangle(\mathcal{N}[\![e_1 : \tau' \to \tau]\!], \mathcal{N}[\![e_2 : \tau']\!]) \\
&\quad = \psi \circ \langle \eta \circ \mathcal{N}[\![e_1 : \tau' \to \tau]\!], [\![t_1]\!] \circ \eta \circ y\rangle \\
&\quad = [\![t_1]\!] \circ \eta \circ \langle \mathcal{N}[\![e_1 : \tau' \to \tau]\!], y\rangle
\end{aligned}$$

Furthermore,

$$\begin{aligned}
&\mathbf{apply}(\pi_1, \eta \circ \pi_2) \circ \langle \mathcal{N}[\![e_1 : \tau' \to \tau]\!], y\rangle \\
&\quad = \mathbf{apply}(\mathcal{N}[\![e_1 : \tau' \to \tau]\!], \eta \circ y) \\
&\quad = \mathcal{N}[\![e_1(v_2) : \tau]\!] \\
&\quad = [\![t_2]\!] \circ \mathcal{N}[\![v : \tau]\!]
\end{aligned}$$

Therefore

$$\begin{aligned}
&\mathcal{C}^{\mathrm{N}}[\![\texttt{vapp}]\!] \circ \langle\rangle(\mathcal{N}[\![e_1 : \tau' \to \tau]\!], \mathcal{N}[\![e_2 : \tau']\!]) \\
&\quad = (\mathbf{apply}(\pi_1, \eta \circ \pi_2))^* \circ [\![t_1]\!] \circ \eta \circ \langle \mathcal{N}[\![e_1 : \tau' \to \tau]\!], y\rangle \\
&\quad = [\![t_1]\!] \circ \mathbf{apply}(\pi_1, \eta \circ \pi_2) \circ \langle \mathcal{N}[\![e_1 : \tau' \to \tau]\!], y\rangle \\
&\quad = [\![t_2 + t_1]\!] \circ \mathcal{N}[\![v : \tau]\!]
\end{aligned}$$

so `vapp` is sound.

To evaluate `let`, if $\mathcal{N}[\![\Gamma \vdash e_1 : \tau']\!] = \bot$, then

$$\begin{aligned}
\mathcal{N}[\![\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau]\!] &= (\mathbf{apply}(\pi_1, \eta \circ \pi_2))^* \circ \psi \circ \langle \eta \circ \mathcal{N}[\![\Gamma \vdash \texttt{lam}\ x.e_2 : \tau' \to \tau]\!], \bot\rangle \\
&= \bot
\end{aligned}$$

as expected, while if $\mathcal{N}[\![\Gamma \vdash e_1 : \tau']\!] = [\![t_1]\!] \circ \eta \circ g_1$, then

$$\begin{aligned}\mathcal{N}[\![\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau]\!] &= [\![t_1]\!] \circ \mathbf{apply}(\mathcal{N}[\![\Gamma \vdash \mathtt{lam}\ x.e_2 : \tau' \to \tau]\!], \eta \circ g_1) \\ &= [\![t_{\mathrm{app}} + t_1]\!] \circ \mathcal{N}[\![\Gamma, x : \tau' \vdash e_2 : \tau]\!] \circ \langle \mathsf{id}, \eta \circ g_1 \rangle\end{aligned}$$

We now rewrite the length program as follows:

```
length2 = rec len.lam l.let l1=l in
                 if (nullp(l1)) then 0 else 1 + len(tail(l1))
```

It should be clear that, as before, $\mathbf{apply}(\mathcal{N}[\![\mathtt{length2}]\!], \perp) = \perp$. Similarly, $\mathcal{N}[\![\mathtt{length2}]\!]$ applied to lists of the form $[\langle z_1, t_1\rangle, \ldots, \langle z_n, \perp\rangle]$ and $[\langle z_1, t_1\rangle, \ldots]$ is again $\perp$.

However,

$$\mathbf{apply}(\mathcal{N}[\![\mathtt{length2}]\!], [\![t]\!] \circ \eta \circ [\,]) = [\![t_{\mathrm{true}} + t + t_{\mathrm{nil?}} + 2t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{n}_0{}^I$$

and

$$\begin{aligned}&\mathbf{apply}(\mathcal{N}[\![\mathtt{length2}]\!], [\![t]\!] \circ \eta \circ [\langle z_1, t_1\rangle, \ldots, \langle z_n, t_n\rangle]) \\ &\quad = [\![t_{\mathrm{false}} + t + t_{\mathrm{nil?}} + 2t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathcal{C}^{\mathrm{N}}[\![+]\!] \\ &\qquad \circ \langle\rangle(\eta \circ \mathsf{n}_1{}^I, \mathbf{apply}(\mathcal{N}[\![\mathtt{length2}]\!], [\![t_1 + t_{\mathrm{tail}}]\!] \circ \eta \circ [\langle z_2, t_2\rangle, \ldots, \langle z_n, t_n\rangle]))\end{aligned}$$

The extra cost of t_{app} comes from the extra binding; however, the cost of taking the tail of the list no longer includes t. Therefore if we assume that

$$\mathbf{apply}(\mathcal{N}[\![\mathtt{length2}]\!], [\![t]\!] \circ \eta \circ [\langle z_1, t_1\rangle, \ldots, \langle z_n, t_n\rangle]) = [\![T(n, t, t_1, \ldots, t_n)]\!] \circ \eta \circ \mathsf{n}_{N(n)}{}^I$$

then we know that $N(n) = n$ as before and that T satisfies the following equations:

$$\begin{aligned}T(0, t) &= t_{\mathrm{true}} + t + t_{\mathrm{nil?}} + 2t_{\mathrm{app}} + t_{\mathrm{rec}} \\ T(n, t, t_1, \ldots, t_n) &= t_+ + t_{\mathrm{false}} + t + t_{\mathrm{nil?}} + 2t_{\mathrm{app}} + t_{\mathrm{rec}} + T(n-1, t_1 + t_{\mathrm{tail}}, t_2, \ldots, t_n)\end{aligned}$$

The solution to T,

$$T(n, t, t_1, \ldots, t_n) = t + (n+1)(t_{\mathrm{nil?}} + 2t_{\mathrm{app}} + t_{\mathrm{rec}}) + n(t_{\mathrm{false}} + t_+ + t_{\mathrm{tail}}) + t_{\mathrm{true}} + \sum_{i=1}^{n} t_i$$

is now $\mathcal{O}(n)$ as in the call-by-value version. Furthermore, the costs embedded in the list are now evaluated only once.

4.5.4 The program tabulate revisited

In Chapter 3 we examined the program tabulate, defined as follows:

$$\begin{aligned}\mathtt{itab} &= \mathtt{rec\ t.lam\ i.lam\ f.lam\ n.if\ i} \leq n\ \mathtt{then\ (f(i))::(t\ (i+1)\ f\ n)\ else\ nil} \\ \mathtt{tabulate} &= \mathtt{itab}(1)\end{aligned}$$

This version of tabulate includes extra code to ensure that the list has a finite length. In this chapter infinite lists are allowed. Therefore let

$$\begin{aligned}\mathtt{ntab} &= \mathtt{rec\ tb.lam\ i.lam\ f.(f(i))::(tb(i+1)f)} \\ \mathtt{ntabulate} &= \mathtt{ntab}(1)\end{aligned}$$

This function always creates an infinite list whose elements are f applied to successively higher integers. Let $z_f : \mathbf{1} \to C\mathcal{T}^{\mathrm{N}}[\![\mathbf{nat} \to \tau]\!]$. Then for any cost t and integer n,

$$\begin{aligned}&\mathbf{apply}(\mathcal{N}[\![\mathtt{ntab}]\!], [\![t]\!] \circ \eta \circ \mathsf{n}_n{}^I, z_f)\\ &\quad = [\![2t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{lcons}^I\\ &\qquad \circ \langle \mathbf{apply}(z_f, [\![t]\!] \circ \eta \circ \mathsf{n}_n{}^I), \mathbf{apply}(\mathcal{N}[\![\mathtt{ntab}]\!], [\![t + t_+]\!] \circ \eta \circ \mathsf{n}_{n+1}{}^I, z_f)\rangle\end{aligned}$$

Thus the function itself is constant-time, but it creates an infinite list with internal cost. The cost of the head of the list is the cost of applying z_f to n which itself has a cost of t. The cost of the integer at the next level then becomes $t + t_+$. Therefore we know that

$$\mathbf{apply}(\mathcal{N}[\![\mathtt{ntabulate}]\!], z_f) = [\![2t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ [\langle z_1, 2t_{\mathrm{app}} + t_{\mathrm{rec}}\rangle, \langle z_2, 2t_{\mathrm{app}} + t_{\mathrm{rec}}\rangle \ldots]$$

where for $i > 0$, $z_i = \mathbf{apply}(z_f, [\![(i-1)t_+]\!] \circ \eta \circ \mathsf{n}_i{}^I)$.

Assuming that z_f is not independent of its input, we would like to reduce the accumulation of t_+ at each element. This can be done by using `let` to evaluate the integer before creating the list. Then z_f is only applied to values, while the cost of adding the integers is moved from the application of z_f to the evaluation of the the tail. Let

$$\begin{aligned}\mathtt{ntab}' &= \mathtt{rec\ tb.lam\ i.lam\ f.let\ j = i\ in\ (f(j))::(tb(j+1)f)}\\ \mathtt{ntabulate}' &= \mathtt{ntab}'(1)\end{aligned}$$

With this program, we have that

$$\mathbf{apply}(\mathcal{N}[\![\mathtt{ntabulate}']\!], z_f) = [\![3t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ [\langle z_1', 3t_{\mathrm{app}} + t_+ + t_{\mathrm{rec}}\rangle, \langle z_2', 3t_{\mathrm{app}} + t_+ + t_{\mathrm{rec}}\rangle \ldots]$$

Where for $i > 0$, $z_i' = \mathbf{apply}(z_f, \eta \circ \mathsf{n}_i{}^I)$. The extra cost of t_{app} comes from the extra variable binding due to the `let` construction. Thus we have changed form a form where the i'th element has an additional cost of $(i-1)t_+$ to a form where each element has the usual cost, but calculating the tail is slightly more expensive.

4.6 Relating call-by-name programs

As with the call-by-value semantics, we can take an ordering $\preceq$ on costs and extend it to an improvement relation $\preceq_\tau^{\mathrm{N}}$ on programs. The process of defining $\preceq_\tau^{\mathrm{N}}$ is very similar to that of defining $\preceq_\tau^{\mathrm{V}}$; the primary difference is that a greater number of subparts contain external cost than in the call-by-value semantics.

The operational definition is essentially the same: Let $e_1 \trianglelefteq^{\mathrm{N}} e_2$ hold when for all contexts $C[\,]$ such that $C[e_1]$ and $C[e_2]$ are closed expressions of ground type, then $C[e_1] \overset{t_1}{\Rightarrow}_{\mathrm{n}} v$ if and only if $C[e_2] \overset{t_2}{\Rightarrow}_{\mathrm{n}} v$ with $t_1 \preceq t_2$.

The definition of the denotational relation $\preceq_\tau^{\mathrm{N}}$ is also very similar. It requires that the meaning of expressions be *monotone* relative to $\preceq$. Therefore, along with the definition of $\preceq_\tau^{\mathrm{N}}$, we use an extension of the definition of monotone to morphisms to the call-by-name meanings of types. The definitions are mutually recursive.

Definition 4.6.1 A morphism $z : \mathbf{1} \to C\mathcal{T}^{\mathrm{N}}[\![\tau]\!]$ is *monotone* relative to a cost order $\preceq$ if either $z = \bot$, or, for some cost t, $z = [\![t]\!] \circ \eta \circ y$, where y is monotone relative to $\preceq$ if one of the following holds:

- τ is a ground type
- $\tau = \delta(\tau_1, \ldots, \tau_n)$, and for all intensional projections $\rho^{\mathrm{N}}_{i_j}$, $\mathsf{id}^* \circ \rho^{\mathrm{N}}_{i_j} \circ y$ is monotone relative to $\preceq$.
- $\tau = \tau_1 \to \tau_2$ and for all monotone $z_1, z_2 : \mathbf{1} \to C\mathcal{T}^{\mathrm{N}}[\![\tau_1]\!]$, if $\mathbf{apply}(\eta \circ y, z_1)$ and $\mathbf{apply}(\eta \circ y, z_2)$ are monotone relative to $\preceq$ and $z_1 \preceq^{\mathrm{N}}_{\tau_1} z_2$ then

$$\mathbf{apply}(\eta \circ y, z_1) \preceq^{\mathrm{N}}_{\tau_2} \mathbf{apply}(\eta \circ y, z_2)$$

When the cost order $\preceq$ is understood, we simply say that z is monotone.

Definition 4.6.2 Given two morphisms $y_1, y_2 : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}[\![\tau]\!]$ that are monotone relative to $\preceq$, $y_1 \preceq^{\mathrm{N}}_{\tau} y_2$ if

- τ is a ground type and $y_1 = y_2$.
- $\tau = \delta(\tau_1, \ldots, \tau_n)$, $F_\delta(!, \ldots, !) \circ y_1 = F_\delta(!, \ldots, !) \circ y_2$, and, for each projection $\rho^{\mathrm{N}}_{i_j}$,

$$\mathsf{id}^* \circ \rho^{\mathrm{N}}_{i_j} \circ y_1 \preceq^{\mathrm{N}}_{\tau_i} \mathsf{id}^* \circ \rho^{\mathrm{N}}_{i_j} \circ y_2$$

- $\tau = \tau_1 \to \tau_2$, and, for all monotone morphisms $z : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}[\![\tau_1]\!]$,

$$\mathbf{apply}(\eta \circ y_1, z) \preceq^{\mathrm{N}}_{\tau_2} \mathbf{apply}(\eta \circ y_2, z)$$

For any monotone morphisms $z_1, z_2 : \mathbf{1} \to C\mathcal{T}^{\mathrm{N}}[\![\tau]\!]$, $z_1 \preceq^{\mathrm{V}}_{\tau_i} z_2$ if $\mathsf{val}(z_1)$ exists precisely when $\mathsf{val}(z_2)$ exists, and, when they both exist, $\mathsf{cost}(z_1) \preceq \mathsf{cost}(z_2)$, and $\mathsf{val}(z_1) \preceq^{\mathrm{N}}_{\tau} \mathsf{val}(z_2)$.

For any type environment Γ, a morphism $\langle\rangle(z_1, \ldots, z_n) : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}[\![\Gamma]\!]$ is monotone if each z_i is monotone. A morphism $g : \mathcal{T}^{\mathrm{N}}[\![\Gamma]\!] \to \mathcal{T}^{\mathrm{N}}[\![\tau]\!]$ is monotone if for all monotone $r, r' : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}[\![\Gamma]\!]$, $g \circ r$ is monotone and if whenever $r \preceq^{\mathrm{N}}_{\Gamma} r'$, $g \circ r \preceq^{\mathrm{N}}_{\tau} g \circ r'$.

Similarly, for any type environment Γ and morphisms $\langle\rangle(z_1, \ldots, z_n), \langle\rangle(z'_1, \ldots, z'_n) : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}[\![\Gamma]\!]$, we say that $\langle\rangle(z_1, \ldots, z_n) \preceq^{\mathrm{N}}_{\Gamma} \langle\rangle(z'_1, \ldots, z'_n)$ if, for each $1 \le i \le n$, $z_i \preceq^{\mathrm{N}}_{\tau_i} z'_i$. Given morphisms $g_1, g_2 : \mathcal{T}^{\mathrm{N}}[\![\Gamma]\!] \to \mathcal{T}^{\mathrm{N}}[\![\tau]\!]$, $g_1 \preceq^{\mathrm{N}}_{\tau} g_2$ if, for all monotone $r : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}[\![\Gamma]\!]$, $g_1 \circ r \preceq^{\mathrm{N}}_{\tau} g_2 \circ r$.

It should be clear that $\preceq^{\mathrm{N}}_{\tau}$ is a pre-order and that for any $t \in T$, $z \preceq^{\mathrm{N}}_{\tau} z'$ if and only if $[\![t]\!] \circ z \preceq^{\mathrm{N}}_{\tau} [\![t]\!] \circ z'$.

We can show that the improvement relation $\preceq^{\mathrm{N}}_{\tau}$ is reasonable by showing that for any closed expressions e_1, e_2 of type τ, $\mathcal{N}_{\mathrm{E}}[\![e_1]\!] \preceq^{\mathrm{N}}_{\tau} \mathcal{N}_{\mathrm{E}}[\![e_2]\!]$ implies that $e_1 \trianglelefteq^{\mathrm{N}} e_2$. Again, to show this, we need to prove that the meanings of all expressions are monotone, and that for any context $C[\ ]$, if $C[e_1]$ and $C[e_2]$ are both closed expressions of type τ', then $\mathcal{N}_{\mathrm{E}}[\![e_1]\!] \preceq^{\mathrm{N}}_{\tau} \mathcal{N}_{\mathrm{E}}[\![e_2]\!]$ implies that $\mathcal{N}_{\mathrm{E}}[\![C[e_1]]\!] \preceq^{\mathrm{N}}_{\tau'} \mathcal{N}_{\mathrm{E}}[\![C[e_2]]\!]$.

Lemma 4.6.1 *Let $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ be a type environment and suppose that $\Gamma \vdash e : \tau$. Then $\mathcal{N}[\![\Gamma \vdash e : \tau]\!]$ is monotone.*

Proof. By induction on e. The proof is straightforward and almost identical to the proof for the call-by-value semantics. $\square$

Theorem 4.6.2 *Suppose that for closed expressions e_1 and e_2 of type τ, $\mathcal{N}[\![e_1 : \tau]\!] \preceq^{\mathrm{N}}_{\tau} \mathcal{N}[\![e_2 : \tau]\!]$. Then for all contexts $C[\,]$, type environments Γ, and types τ' such that both $\Gamma \vdash C[e_1] : \tau'$ and $\Gamma \vdash C[e_2] : \tau'$,*

$$\mathcal{N}[\![\Gamma \vdash C[e_1] : \tau']\!] \preceq^{\mathrm{N}}_{\tau'} \mathcal{N}[\![\Gamma \vdash C[e_2] : \tau']\!]$$

Proof. By induction on the structure of $C[\,]$. The proof is straightforward and also similar to the proof for the call-by-value semantics. We include the case for application here.

Let $r : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}[\![\Gamma]\!]$. Then $\mathcal{N}[\![\Gamma \vdash C[e_1] : \tau']\!] \preceq^{\mathrm{N}}_{\tau'} \mathcal{N}[\![\Gamma \vdash C[e_2] : \tau']\!]$ if

$$\mathcal{N}[\![\Gamma \vdash C[e_1] : \tau']\!] \circ r \preceq^{\mathrm{N}}_{\tau'} \mathcal{N}[\![\Gamma \vdash C[e_2] : \tau']\!] \circ r$$

For application assume that $C[\,] = C_1[\,](C_2[\,])$. Even though only one of C_1 or C_2 may actually have a hole, we can ignore this fact during the proof, because if C_i does not have a hole, then $C_i[e_1] \preceq^{\mathrm{N}}_{\tau''} C_i[e_2]$ holds immediately.

Because $\Gamma \vdash C_1[e_1](C_2[e_1]) : \tau'$ and $\Gamma \vdash C_1[e_2](C_2[e_2]) : \tau'$, there exist a type τ'' such that $\Gamma \vdash C_1[e_1] : \tau'' \to \tau'$, $\Gamma \vdash C_2[e_1] : \tau''$, $\Gamma \vdash C_1[e_2] : \tau'' \to \tau'$, and $\Gamma \vdash C_2[e_2] : \tau''$. Therefore by the induction hypothesis, $\mathcal{N}[\![\Gamma \vdash C_1[e_1] : \tau'' \to \tau']\!] \circ r \preceq^{\mathrm{N}}_{\tau'' \to \tau'} \mathcal{N}[\![\Gamma \vdash C_1[e_2] : \tau'' \to \tau']\!] \circ r$ and $\mathcal{N}[\![\Gamma \vdash C_2[e_1] : \tau'']\!] \circ r \preceq^{\mathrm{N}}_{\tau''} \mathcal{N}[\![\Gamma \vdash C_2[e_2] : \tau'']\!] \circ r$. By Lemma 4.6.1, $\mathcal{N}[\![\Gamma \vdash C_2[e_1] : \tau'']\!] \circ r$, $\mathcal{N}[\![\Gamma \vdash C_2[e_2] : \tau'']\!] \circ r$, $\mathcal{N}[\![\Gamma \vdash C_1[e_1] : \tau'' \to \tau']\!] \circ r$, and $\mathcal{N}[\![\Gamma \vdash C_1[e_2] : \tau'' \to \tau']\!] \circ r$ are all monotone. Therefore by the definition of $\preceq^{\mathrm{N}}_{\tau'' \to \tau'}$,

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{N}[\![\Gamma \vdash C_1[e_1](C_2[e_1]) : \tau']\!] \circ r} \\ = & \mathbf{apply}(\mathcal{N}[\![\Gamma \vdash C_1[e_1] : \tau'' \to \tau']\!] \circ r, \mathcal{N}[\![\Gamma \vdash C_2[e_1] : \tau'']\!] \circ r) \\ \preceq^{\mathrm{N}}_{\tau'} & \mathbf{apply}(\mathcal{N}[\![\Gamma \vdash C_1[e_1] : \tau'' \to \tau']\!] \circ r, \mathcal{N}[\![\Gamma \vdash C_2[e_2] : \tau'']\!] \circ r) \\ \preceq^{\mathrm{N}}_{\tau'} & \mathbf{apply}(\mathcal{N}[\![\Gamma \vdash C_1[e_2] : \tau'' \to \tau']\!] \circ r, \mathcal{N}[\![\Gamma \vdash C_2[e_2] : \tau'']\!] \circ r) \\ = & \mathcal{N}[\![\Gamma \vdash C_1[e_2](C_2[e_2]) : \tau']\!] \circ r \end{array}$$

□

Theorem 4.6.3 *Suppose that for any ground type g and closed values v, v' of type g, $\mathcal{N}[\![v : g]\!] = \mathcal{N}[\![v' : g]\!]$ implies that $v = v'$. Then for any closed expressions e_1 and e_2 of type τ. If $\mathcal{N}[\![e_1]\!] \preceq^{\mathrm{N}}_{\tau} \mathcal{N}[\![e_2]\!]$, then $e_1 \trianglelefteq^{\mathrm{N}} e_2$.*

Proof. Let $C[\,]$ be a context such that $C[e_1]$ and $C[e_2]$ are closed values of ground type g. Then by Theorem 4.6.2 $\mathcal{N}[\![C[e_1]]\!] \preceq^{\mathrm{N}}_{g} \mathcal{N}[\![C[e_2]]\!]$. This means that if $C[e_1] \overset{t_1}{\Rightarrow}_{\mathrm{n}} v$, then $\mathcal{N}[\![C[e_1]]\!] = [\![t_1]\!] \circ \mathcal{N}[\![v]\!]$. Thus, for some cost t_2, $\mathcal{N}[\![C[e_2]]\!] = [\![t_2]\!] \circ \mathcal{N}[\![v]\!]$ as well, with $t_1 \preceq t_2$. From adequacy we know that there must exist a value v' such that $C[e_2] \overset{t_2}{\Rightarrow}_{\mathrm{n}} v'$; this means, however, that $\mathcal{N}[\![v']\!] = \mathcal{N}[\![v]\!]$, so $v = v'$. Therefore $e_1 \trianglelefteq^{\mathrm{N}} e_2$. □

4.7 Conclusion

The derivation of the intensional call-by-name semantics follows the same pattern as the derivation of the intensional call-by-value semantics. The differences between them are governed primarily by the differences between the extensional semantics. Thus it is not surprising that the call-by-name denotational semantics proves to be sound and adequate relative to the call-by-name operational semantics, and that the proofs of soundness and adequacy strongly resemble the proofs of soundness and adequacy for the call-by-value semantics.

Similarly, when relating programs, the definition of $\preceq^{\mathrm{N}}_{\tau}$ strongly resembles the definition of $\preceq^{\mathrm{V}}_{\tau}$, differing only in the way it handles the extra cost in subparts. Thus proving that expressions were monotone and that $\preceq^{\mathrm{N}}_{\tau}$ implied $\trianglelefteq^{\mathrm{N}}$ is greatly simplified, as the proofs were almost identical to the equivalent proofs for the call-by-value definitions.

The uniformity between the call-by-name and call-by-value semantics is also reflected in the treatment of delayed computations. With a call-by-value evaluation strategy, the only computations that are delayed are those stored in functional types. With cost structures we are able to treat the delay due to higher order types and the delay due to lazy structures uniformly. Thus there is no difficulty handling the delay in the structures such as lazy lists while still maintaining the ability to handle higher order types. This differs from the approach of [37], where the addition of higher-order types and lazy structures were handled in substantially different ways, making it difficult to include both.

When calculating actual examples, we see quickly that limiting ourselves to only call-by-name application can lead to substantial inefficiency. Call-by-name languages such as Algol solve this problem by introducing a call-by-value `let` constructs, and such a construct does solve the problems with inefficiency if used appropriately. Other languages that use macros in a manner similar to call-by-name, such as macros in C, do not allow recursion, limiting the degree of inefficiency introduced.

In practice, call-by-name languages are rarely used by themselves; instead, they include some methods for avoiding reevaluations – either with a `let` construct or by using a call-by-need evaluation strategy. As a call-by-need strategy has the same extensional effect (when we know that the value of an expression cannot change, something that is *not* true with Algol-60), it is generally safe, and sometimes more efficient, to use it rather than a straight call-by-name strategy. It is not always clear how the cost calculated with the call-by-name semantic relates to actual costs using a call-by-need evaluation strategy. It is true, however, that, ignoring any extra overhead, the cost of running a program using a call-by-need evaluation strategy will be less than running a program using a call-by-name evaluation strategy. Thus many times analyzing a program with the call-by-name intensional semantics will give us an upper bound on the cost of running a program using a call-by-need evaluation strategy. Sometimes, when we know that little or no reevaluation takes place, the bound may be a very good one; for example, the call-by-need cost of `ntabulate`′ should be very close to the cost found in section 4.5.

Chapter 5

A Larger Example: Pattern Matching

5.1 Regular expression search

When evaluating by call-by-value, we can generally determine the cost or complexity of an expression easily when the expression does not contain any higher-order elements. For example, when we examined the `length` function in Chapter 3 we were easily able to determine the cost function; the power of the full semantics was not needed to determine it. We now turn to a more interesting example involving higher-order functions.

The higher-order functions examined so far all used functions as inputs which were then applied a number of times. In practice, many high-order functions use their arguments in this manner. The example in this chapter, however, builds and applies higher-order functions within it; the outer function uses only values of first-order types as inputs. These higher-order functions are called *continuations* because they describe the future behavior of the overall function, i.e., the action a function needs to take when it wants to continue processing information. Thus the main function takes, as one of its arguments, a continuation, does some internal calculation, then, depending on the result, calls the continuation or a modified continuation on a suitable new argument to determine the final value.

The example in this section is a function, `accept`, that takes as input a regular expression and a string and returns `true` if the string is in the language described by the regular expression and `false` otherwise. For the purposes of discussion, we will use the following notation for the regular expressions, where the letters r and q indicate regular expressions, and a indicates a single character in some set I. If R and Q are sets of strings, then we write RQ for the set of strings obtained by concatenation, that is, $RQ = \{s_1 s_2 \mid s_1 \in R \text{ and } s_2 \in Q\}$. This notation generalizes to R^i $(i \geq 0)$, the set of strings obtained by concatenating i strings from R. Finally let L_r be the set of strings matching r, defined inductively:

a	single character	$L_a = \{a\}$
rq	sequencing	$L_{rq} = L_r L_q$
$r\|q$	choice	$L_{r\|q} = L_r \cup L_q$
r^*	repetition	$L_{r^*} = \bigcup_{i=0}^{\infty} L_r^i$

We assume that repetition has higher precedence than sequencing, which has higher precedence than choice; thus $a|ab^*$ is equivalent to $a|(a(b^*))$. We also assume that sequencing and choice associate to the right; for example, $abc = a(bc)$. This is not the standard association, but as sequencing and choice are associative operators (i.e, $L_{(ab)c} = L_{a(bc)}$ and $L_{(a|b)|c} = L_{a|(b|c)}$) this decision is largely immaterial, except that with this choice many of the complexity calculations become significantly simpler, as will be seen in Example 1-3 of section 5.1.5. Examples 7 and 8 of section 5.1.7. Lastly, we write $[a_1 a_2 \ldots a_n]$ as shorthand for $(a_1|a_2|...|a_n)$.

There is a standard algorithm ([2]) used in many applications of regular expression matching that has a worst-case complexity proportional to the product of the length of the string being matched and the size of the regular expression, and is linear in the length of the string in general. This algorithm, however, does a non-trivial amount of pre-processing work initially. Therefore for cases where the regular expressions are simple and frequently changing (as in incremental searches by an editor), the matching algorithm given in this section may be faster.

5.1.1 The algorithm

The algorithm consists of two primary parts: an internal recursive function, `acc`, taking a continuation as an argument, and an external function, `accept`, that calls `acc` with a final continuation which would be called whenever `acc` reaches the end of the string to be searched. The function `acc` has three arguments: the regular expression (r), the string to match (s), and the continuation (k). Overall, `acc` r k s returns true if there exist strings s_1 and s_2, such that $s = s_1 s_2$, r matches s_1, and $k(s_2)$ returns true. The execution of `acc` r k s is as follows:

$r = a$: If the first character of s is a, then call k on the rest of s, otherwise return `false`.

$r = r_1 | r_2$: Call `acc` on r_1, s and k; if the result is `true`, return true, otherwise return the result of calling `acc` on r_2, s, and k.

$r = r_1 r_2$: Build a new continuation that, given a string s', calls `acc` on r_2, k, and s', and call `acc` on r_1, s and the new continuation.

$r = (r')^*$: First apply the continuation to s to see if any copies of r' are needed. If k applied to s is `true`, return true (the empty string matches $(r')^*$). Otherwise, build a new continuation, `strcheck`, which takes a string s' and returns `false` if $s' = s$, otherwise `strcheck` returns the result of applying `acc` to r, k and s'. For the final result return the result of calling `acc` on r', `strcheck` and s. The check that $s = s'$ ensures that the algorithm terminates; if no progress is made by the time the `strcheck` is called, there is no possible successful match.

In ML, the algorithm becomes the following program, using a suitable data type to represent regular expressions:

```
datatype regexp = CHAR of string | OR of regexp * regexp
                | SEQ of regexp * regexp | STAR of regexp

fun acc (CHAR i) k (j::s) = (i = j) andalso (k s)
  | acc (CHAR i) k nil = false
  | acc (OR (r1, r2)) k s = (acc r1 k s) orelse (acc r2 k s)
  | acc (SEQ (r1, r2)) k s = (acc r1 (acc r2 k) s)
  | acc (STAR r1) k s
    = let
           fun strcheck s2 = not (length (s) = length(s2))
                             andalso (acc (STAR r1) k s2)
      in
           (k s) orelse (acc r1 strcheck s)
      end
```

We need only to compare string lengths rather than entire strings for the equality check because `acc (STAR r1) k s` is designed so that if a recursive call of the form `acc (STAR r1)` k' s' occurs `s'` will be a suffix of `s`. Therefore comparing lengths gives the same result as comparing contents, but the calculation can be implemented more efficiently.

The external program calls `acc` with a continuation that returns `true` if its input string is empty (meaning that all characters were matched) and `false` otherwise, i.e., the standard test for an empty string, `null`. In ML it becomes

```
fun accept r s = acc r null s
```

5.1.2 Converting from ML

The next step converts the program from ML to a language consistent with the semantic definitions given in this dissertation. This involves two steps: converting all the types, and converting the pattern-matching format of ML to λ-expressions. We represent characters as integers for simplicity. They could be considered as items themselves, or as ASCII code, it matters little to the analysis of the algorithm. To make the string comparison needed in `strcheck` constant time, we actually pair lists of characters with an integer that corresponds to the length of the string; however, so that we do not have to calculate the length of the string during initialization, we pair a list of characters s with an integer i such that $i = \texttt{length}(s_0) - \texttt{length}(s)$, where s_0 is the "original" string, i.e, the argument of `accept`. Thus i is the number of characters already read since the call to `accept`. A pair of this kind is used to represent a string internally within the body of `acc`.

Lastly, we need a representation for the regular expression type. The constants given in FL are insufficient, so we add a new (ground) type, **regexp**, and new constants. The constructors for **regexp** are as follows:

Name	Arity	Type
`Char`	1	**nat** $\rightarrow$ **regexp**
`Or`	2	**regexp** $\rightarrow$ **regexp** $\rightarrow$ **regexp**
`Seq`	2	**regexp** $\rightarrow$ **regexp** $\rightarrow$ **regexp**
`Star`	1	**regexp** $\rightarrow$ **regexp**

We additionally include one "pattern-matching" primitive function, rcase, of arity 5 such that for any type τ, rcase has type

$$\begin{aligned}&\mathbf{regexp} \to (\mathbf{nat} \to \tau) \to (\mathbf{regexp} \to \mathbf{regexp} \to \tau) \to (\mathbf{regexp} \to \mathbf{regexp} \to \tau)\\ &\qquad \to (\mathbf{regexp} \to \tau) \to \tau\end{aligned}$$

This constant is similar in form to case, except that the different cases for **regexp** are handled instead. For clarity we write

$$\texttt{rcase}\ (e)\ \texttt{of}\ \texttt{CHAR} : e_1 \mid \texttt{OR} : e_2 \mid \texttt{SEQ} : e_3 \mid \texttt{STAR} : e_4$$

to stand for rcase $e\ e_1\ e_2\ e_3\ e_4$. Its operational rules are as follows:

$$\frac{v_1\ n \overset{t}{\Rightarrow}_{\mathrm{v}} v}{\texttt{rcase}\ (\texttt{Char}\ n)\ \texttt{of}\ \texttt{CHAR} : v_1 \mid \texttt{OR} : v_2 \mid \texttt{SEQ} : v_3 \mid \texttt{STAR} : v_4 \overset{t+t_{\mathrm{rcase}}}{\Rightarrow}_{\mathrm{v}} v}$$

$$\frac{v_2\ r_1\ r_2 \overset{t}{\Rightarrow}_{\mathrm{v}} v}{\texttt{rcase}\ (\texttt{Or}\ r_1\ r_2)\ \texttt{of}\ \texttt{CHAR} : v_1 \mid \texttt{OR} : v_2 \mid \texttt{SEQ} : v_3 \mid \texttt{STAR} : v_4 \overset{t+t_{\mathrm{rcase}}}{\Rightarrow}_{\mathrm{v}} v}$$

$$\frac{v_3\ r_1\ r_2 \overset{t}{\Rightarrow}_{\mathrm{v}} v}{\texttt{rcase}\ (\texttt{Seq}\ r_1\ r_2)\ \texttt{of}\ \texttt{CHAR} : v_1 \mid \texttt{OR} : v_2 \mid \texttt{SEQ} : v_3 \mid \texttt{STAR} : v_4 \overset{t+t_{\mathrm{rcase}}}{\Rightarrow}_{\mathrm{v}} v}$$

$$\frac{v_4\ r \overset{t}{\Rightarrow}_{\mathrm{v}} v}{\texttt{rcase}\ (\texttt{Star}\ r)\ \texttt{of}\ \texttt{CHAR} : v_1 \mid \texttt{OR} : v_2 \mid \texttt{SEQ} : v_3 \mid \texttt{STAR} : v_4 \overset{t+t_{\mathrm{rcase}}}{\Rightarrow}_{\mathrm{v}} v}$$

We also define some "constants" for strings. These are not real constants, but rather abbreviations which make the final code more understandable and make calculating its meaning easier. For example, snil? is the equivalent of nil? for strings. Similarly, shead returns the first character of the string, while, stail returns the rest of the string, adjusting the number of read characters in the right-hand side of the pair. The function sseen returns the number of "seen" (or read) characters. If we were using these strings outside of this program, the definition of sseen would be meaningless, but within the program, this definition of sseen will be sufficient and of constant cost. It is used instead of the length program so that we do not need to initially evaluate the length of a string; within the program sseen(s) + length(s) is always the length of the initial string being matched.

Lastly snew creates a new string from a list of integers. The abbreviations are thus as follows:

```
string  = list(nat) × nat
snil?   = lam s.nil?(fst s)
shead   = lam s.head(fst s)
stail   = lam s.<tail(fst s), (snd s) + 1̅>
sseen   = snd
snew    = lam l.<l, 0̅>
```

Besides adding a type for regular expressions, the ML version used both pattern matching and the constants andalso, orelse, and not, none of which are in our language. The pattern matching is replaced by snil? and rcase. For the boolean constants we can easily assign simple abbreviations: Let e_1 andalso e_2 stand for if e_1 then e_2 else false, and let e_1 orelse e_2 stand for if e_1 then true else e_2. Expanding not in the case where it is used would lead to a less efficient program; instead we simply write `if` e_1 `then false else` e_2 for (`not` e_1) `andalso` e_2. We can now translate each case from the ML version to an (unclosed) expression. Let the code fragments `accchar`, `accor`, `accseq`, `accstar`, and `strcheck` be defined by

```
accchar   =def  lam n.if (snil? s) then false
                       else (shead(s) = n) andalso (stail s)
accor     =def  lam r1.lam r2.(a r1 k s) orelse (a r2 k s)
accseq    =def  lam r1.lam r2.a r1 (a r2 k) s
accstar   =def  lam r1.(k s) orelse (a r1 strcheck s)
strcheck  =def  lam s'.if (sseen(s) = sseen(s')) then false
                       else (a (Star r1) k s')
```

The free variables in the above code fragments are `a`, representing the recursive variable representing `acc`; `s`, representing the current string to check; and `k`, representing the continuation. Additionally, `strcheck` has the free variable `r1` from `accstar`. The main programs are as follows:

```
accept  =def  lam r.lam l.acc r snil? (snew l)
acc     =def  rec a.lam r.lam k.lam s.
                    rcase r of CHAR: accchar | OR: accor
                                | SEQ: accseq | STAR: accstar
```

5.1.3 Denotational semantic definitions for regexp constants

Before we can calculate the meaning of the two programs, we need to give meanings to the new constants. As we will be using lists and integers for strings and characters, we only need to provide meanings in the categories **PDom** and $\mathbf{PDom}^{\rightarrow}$.

In the ML program, the regular expression type was defined as the union of four different types: one containing integers, one containing a regular expression, and two containing pairs of regular expressions. Therefore let R be the least solution to the pre-domain equation

$$R \cong ((\mathbf{N} + (R \times R)) + (R \times R)) + R$$

The syntactic set of regular expressions (taking some care with parentheses), discretely ordered, is a solution to the above equation.

We chose the ordering of the subparts of R to match the definition of `rcase`; the choice of parentheses was arbitrary.

Because $\mathbf{N}$ is discretely ordered and products and sums preserve discreteness in **PDom**, R is also discretely ordered. Let

$$\begin{aligned} \mathsf{rfold}\colon&\ ((\mathbf{N} + (R \times R)) + (R \times R)) + R \rightarrow R \\ \mathsf{runfold}\colon&\ R \rightarrow ((\mathbf{N} + (R \times R)) + (R \times R)) + R \end{aligned}$$

be the derived isomorphisms. We now can shift to the intensional category $\mathbf{PDom}^{\rightarrow}$ by setting R^I to id_R, rfold^I to $(\mathsf{rfold}, \mathsf{rfold})$, and $\mathsf{runfold}^I$ to $(\mathsf{runfold}, \mathsf{runfold})$. We next need some morphisms equivalent to the constructors. These simply insert an element into the proper place in the above sum and then use rfold to convert the sum to R. Thus let rchar, ror, rseq, and rstar be defined as

$$
\begin{array}{ll}
\mathcal{C}^{\mathrm{V}}[\![\texttt{Char}]\!]: & \times(\mathbf{N}^I) \to CR^I \\
\mathcal{C}^{\mathrm{V}}[\![\texttt{Or}]\!]: & \times(R^I, R^I) \to CR^I \\
\mathcal{C}^{\mathrm{V}}[\![\texttt{Seq}]\!]: & \times(R^I, R^I) \to CR^I \\
\mathcal{C}^{\mathrm{V}}[\![\texttt{Star}]\!]: & \times(R^I) \to CR^I \\
\mathcal{C}^{\mathrm{V}}[\![\texttt{rcase}]\!] & \times(R^I, [\mathbf{N}^I \Rightarrow CR^I], [R^I \Rightarrow C[R^I \Rightarrow CR^I]], [R^I \Rightarrow C[R^I \Rightarrow CR^I]], [R^I \Rightarrow CR^I]) \to CR^I
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{C}^{\mathrm{V}}[\![\texttt{Char}]\!] & = & \eta \circ \mathsf{rchar} \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}[\![\texttt{Or}]\!] & = & \eta \circ \mathsf{ror} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}[\![\texttt{Seq}]\!] & = & \eta \circ \mathsf{rseq} \circ (\pi_2 \times \mathsf{id}) \\
\mathcal{C}^{\mathrm{V}}[\![\texttt{Star}]\!] & = & \eta \circ \mathsf{rstar} \circ \pi_2 \\
\mathcal{C}^{\mathrm{V}}[\![\texttt{rcase}]\!]: & = & [\![t_{\mathrm{rcase}}]\!] \circ \mathsf{rcase}(\mathsf{app1}_{\mathbf{N}^I} \circ (\mathsf{id} \times \pi_1^4), \mathsf{app2} \circ (\mathsf{id} \times \pi_2^4), \\
 & & \qquad \mathsf{app2} \circ (\mathsf{id} \times \pi_3^4), \mathsf{app1}_{R^I} \circ (\mathsf{id} \times \pi_4^4)) \circ \chi
\end{array}
$$

where $\mathsf{app1}_A : A \times [A \Rightarrow CR^I] \to CR^I$ and $\mathsf{app2} : (R^I \times R^I) \times [R^I \Rightarrow C[R^I \Rightarrow R^I]] \to CR^I$ such that $\mathsf{app1} = \mathsf{app} \circ \beta$ and $\mathsf{app2} = \mathsf{app}^* \circ \psi \circ (\mathsf{app} \times \eta) \circ \alpha^l \circ \beta$, and χ is the product isomorphism from $\times(R^I, A, B, C, D)$ to $R^I \times (\times(A, B, C, D))$.

Figure 5.1: Semantic definitions of regular expression constants

follows:

$$
\begin{array}{rcl}
\mathsf{rchar} : \mathbf{N}^I \to R^I & = & \mathsf{rfold}^I \circ \mathsf{inl} \circ \mathsf{inl} \circ \mathsf{inl} \\
\mathsf{ror} : R^I \times R^I \to R^I & = & \mathsf{rfold}^I \circ \mathsf{inl} \circ \mathsf{inl} \circ \mathsf{inr} \\
\mathsf{rseq} : R^I \times R^I \to R^I & = & \mathsf{rfold}^I \circ \mathsf{inl} \circ \mathsf{inr} \\
\mathsf{rstar} : R^I \to R^I & = & \mathsf{rfold}^I \circ \mathsf{inr}
\end{array}
$$

These four morphisms are the equivalents of inl and inr for the four-part sum. Therefore we will also need a four-part equivalent to case. Suppose there exist objects p, p', and morphisms $f_1 : \mathbf{N}^I \times p \to p'$, $f_2, f_3 : (R^I \times R^I) \times p \to p'$, and $f_4 : R^I \times p \to p'$. Each morphism has as input an element of a regular expression and some "other" argument. In our case the extra argument contains the function to apply to the regular expression elements. We can thus let rcase be shorthand for the multiple case check by setting $\mathsf{rcase}(f_1, f_2, f_3, f_4)$ to be

$$\mathsf{rcase}(f_1, f_2, f_3, f_4) : R^I \times p \to p' = \mathsf{case}(\mathsf{case}(\mathsf{case}(f_1, f_2), f_3), f_4) \circ (\mathsf{runfold}^I \times \mathsf{id})$$

The only properties we need for rchar, ror, rseq, rstar are the following:

Lemma 5.1.1 *For any $f_1 : \mathbf{N}^I \times p \to p'$, f_2, $f_3 : (R^I \times R^I) \times p \to p'$, and $f_4 : R^I \times p \to p'$,*

$$
\begin{array}{rcl}
\mathsf{rcase}(f_1, f_2, f_3, f_4) \circ (\mathsf{rchar} \times \mathsf{id}) & = & f_1 \\
\mathsf{rcase}(f_1, f_2, f_3, f_4) \circ (\mathsf{ror} \times \mathsf{id}) & = & f_2 \\
\mathsf{rcase}(f_1, f_2, f_3, f_4) \circ (\mathsf{rseq} \times \mathsf{id}) & = & f_3, \text{and} \\
\mathsf{rcase}(f_1, f_2, f_3, f_4) \circ (\mathsf{rstar} \times \mathsf{id}) & = & f_4
\end{array}
$$

The meanings for the **regexp** constants are listed in Figure 5.1. The meanings of the constructors are formed by applying a product isomorphism to convert the input from the $\times(A_1, \ldots, A_n)$ form, applying the appropriate constructor morphism defined earlier, and lastly adding trivial cost via η. For `rcase`, $\mathsf{app1}$ takes input of the form $\langle a, f \rangle$ and returns the result of applying f to a. $\mathsf{app2}$ is similar, except that it takes a triple $\langle\langle a_1, a_2\rangle, f\rangle$ and applies f first to a_1 then a_2. Thus the meaning of `rcase` takes a regular expression and four functions, each describing how to handle each type of regular expression, determines the type of regular expression, and applies the resulting element to the relevant function.

As `Char`, `Or`, `Seq`, and `Star` are constructors we immediately know they are sound. For `rcase` it is more complicated:

Theorem 5.1.2 $\mathcal{C}^{\mathrm{V}}[\![\texttt{rcase}]\!]$ *is sound for* `rcase`

Proof. The proof involves checking each of the four operational rules. The proofs are all similar; we prove the case when the operational rule is

$$\frac{v_2\ r_1\ r_2 \overset{t}{\Rightarrow}_{\mathrm{v}} v}{\texttt{rcase}\ (\texttt{Or}\ r_1\ r_2)\ \texttt{of CHAR} : v_1 \mid \texttt{OR} : v_2 \mid \texttt{SEQ} : v_3 \mid \texttt{STAR} : v_4 \overset{t+t_{\mathrm{rcase}}}{\Rightarrow}_{\mathrm{v}} v}$$

By assumption $\texttt{rcase}\ (\texttt{Or}\ r_1\ r_2)\ \texttt{of CHAR} : v_1 \mid \texttt{OR} : v_2 \mid \texttt{SEQ} : v_3 \mid \texttt{STAR} : v_4$ is well typed, therefore we know that $\vdash r_1, r_2 : \mathbf{regexp}$, and there exists a type τ such that

$$\vdash v_1 : \mathbf{nat} \to \tau, \quad \vdash v_2, v_3 : \mathbf{regexp} \to \mathbf{regexp} \to \tau, \text{ and } \quad \vdash: v_4 : \mathbf{regexp} \to \tau$$

Furthermore, r_1, r_2, v_1, v_2, v_3, v_4 are all values, so there exist morphisms $y_1', y_2' : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{regexp}]\!]$, $y_1 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{nat} \to \tau]\!]$, $y_2, y_3 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{regexp} \to \mathbf{regexp} \to \tau]\!]$, and $y_4 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{regexp} \to \tau]\!]$ such that, for $i = 1$, 2, $\mathcal{V}[\![r_i]\!] = \eta \circ y_i'$, and, for $1 \le i \le r$, $\mathcal{V}[\![v_i]\!] = \eta \circ y_i$. This also means that

$$\mathcal{V}[\![\texttt{Or}\ r_1\ r_2 : \mathbf{regexp}]\!] = \eta \circ \mathsf{ror} \circ (\pi_2 \times \mathsf{id}) \circ \langle\rangle(y_1', y_2') = \eta \circ \mathsf{ror} \circ \langle y_1', y_2' \rangle$$

By the definition of soundness, we can assume as an induction hypothesis that for some cost t, $\mathcal{V}[\![v_2\ r_1\ r_2 : \tau]\!] = [\![t]\!] \circ \mathcal{V}[\![v : \tau]\!]$. Therefore

$$\begin{aligned}
&\mathsf{app2} \circ (\mathsf{id} \times \pi_2^4) \circ \langle \langle y_1', y_2' \rangle, \langle\rangle(y_1, y_2, y_3, y_4) \rangle \\
&\quad = \mathsf{app}^* \circ \psi \circ (\mathsf{app} \times \eta) \circ \alpha^l \circ \beta \circ \langle \langle y_1', y_2' \rangle, y_2 \rangle \\
&\quad = \mathsf{app}^* \circ \psi \circ (\mathsf{app} \times \eta) \circ \langle \langle y_2, y_1' \rangle, y_2' \rangle \\
&\quad = \mathsf{app}^* \circ \psi \circ \langle \mathsf{app} \circ \langle y_2, y_1' \rangle, \eta \circ y_2' \rangle \\
&\quad = \mathsf{app}^* \circ \psi \circ \langle \mathcal{V}[\![v_2(r_1) : \mathbf{regexp} \to \tau]\!], \mathcal{V}[\![r_2 : \mathbf{regexp}]\!] \rangle \\
&\quad = \mathcal{V}[\![v_2\ r_1\ r_2 : \tau]\!] \\
&\quad = [\![t]\!] \circ \mathcal{V}[\![v : \tau]\!]
\end{aligned}$$

Thus

$$\begin{aligned}
&\mathcal{V}[\![\texttt{rcase}\ (\texttt{Or}\ r_1\ r_2)\ \texttt{of CHAR} : v_1 \mid \texttt{OR} : v_2 \mid \texttt{SEQ} : v_3 \mid \texttt{STAR} : v_4 : \tau]\!] \circ \langle\rangle(\mathsf{ror} \circ \langle y_1', y_2' \rangle, y_1, y_2, y_3, y_4) \\
&\quad = [\![t_{\mathrm{rcase}}]\!] \circ \mathsf{rcase}(\mathsf{app1} \circ (\mathsf{id} \times \pi_1^4), \mathsf{app2} \circ (\mathsf{id} \times \pi_2^4), \mathsf{app2} \circ (\mathsf{id} \times \pi_3^4), \mathsf{app1} \circ (\mathsf{id} \times \pi_4^4)) \\
&\qquad\qquad \circ \chi \circ \langle\rangle(\mathsf{ror} \circ \langle y_1', y_2' \rangle, y_1, y_2, y_3, y_4) \\
&\quad = [\![t_{\mathrm{rcase}}]\!] \circ \mathsf{rcase}(\mathsf{app1} \circ (\mathsf{id} \times \pi_1^4), \mathsf{app2} \circ (\mathsf{id} \times \pi_2^4), \mathsf{app2} \circ (\mathsf{id} \times \pi_3^4), \mathsf{app1} \circ (\mathsf{id} \times \pi_4^4)) \\
&\qquad\qquad \circ (\mathsf{ror} \times \mathsf{id}) \circ \langle \langle y_1', y_2' \rangle, \langle\rangle(y_1, y_2, y_3, y_4) \rangle \\
&\quad = [\![t_{\mathrm{rcase}}]\!] \circ \mathsf{app2} \circ (\mathsf{id} \times \pi_2^4) \circ \langle \langle y_1', y_2' \rangle, \langle\rangle(y_1, y_2, y_3, y_4) \rangle \\
&\quad = [\![t_{\mathrm{rcase}}]\!] \circ [\![t]\!] \circ \mathcal{V}[\![v : \tau]\!] \\
&\quad = [\![t + t_{\mathrm{rcase}}]\!] \circ \mathcal{V}[\![v : \tau]\!]
\end{aligned}$$

□

Because **regexp** is a ground type, we know by Theorem A.2.1 that all the constants except `rcase` are adequate. It is also straightforward to show that $\mathcal{EC}^{\mathrm{V}}[\![\texttt{rcase}]\!]$ is adequate; we omit the formal proof, although it is similar to the proof that `case` is adequate.

$$
\begin{array}{lcl}
\mathbf{apply}(\mathcal{V}[\![\mathtt{shead}]\!], \langle [\,], \mathsf{n}_k\rangle) & = & \bot \\
\mathbf{apply}(\mathcal{V}[\![\mathtt{shead}]\!], \langle [a_1, \ldots, a_n], \mathsf{n}_k\rangle) & = & [\![t_{\mathrm{app}} + t_{\mathrm{fst}} + t_{\mathrm{head}}]\!] \circ \eta \circ a_1 \\
\mathbf{apply}(\mathcal{V}[\![\mathtt{stail}]\!], \langle [\,], \mathsf{n}_k\rangle) & = & \bot \\
\mathbf{apply}(\mathcal{V}[\![\mathtt{stail}]\!], \langle [a_1, \ldots, a_n], \mathsf{n}_k\rangle) & = & [\![t_{\mathrm{app}} + t_{\mathrm{fst}} + t_{+} + t_{\mathrm{snd}} + t_{\mathrm{tail}}]\!] \circ \eta \circ \langle [a_2, \ldots, a_n], \mathsf{n}_{k+1}\rangle \\
\mathbf{apply}(\mathcal{V}[\![\mathtt{snil?}]\!], \langle [a_1, \ldots, a_n], \mathsf{n}_k\rangle) & = & [\![t_{\mathrm{app}} + t_{\mathrm{fst}} + t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathbf{bool}(n = 0) \\
\mathbf{apply}(\mathcal{V}[\![\mathtt{snew}]\!], [a_1, \ldots, a_n]) & = & [\![t_{\mathrm{app}}]\!] \circ \eta \circ \langle [a_1, \ldots, a_n], \mathsf{n}_0\rangle \\
\mathbf{apply}(\mathcal{V}[\![\mathtt{sseen}]\!], \langle [a_1, \ldots, a_n], \mathsf{n}_k\rangle) & = & [\![t_{\mathrm{snd}}]\!] \circ \eta \circ \mathsf{n}_k
\end{array}
$$

Figure 5.2: Semantic meanings of string functions

5.1.4 Denotational Interpretation of `accept` and `acc`

Before determining the meanings of `accept` and `acc`, we first calculate some of the properties of the string functions. The first five properties are similar to the list properties shown in Figure 2.18 of Chapter 3, except that there is extra cost both because the abbreviations are abstractions, and because one must use the product functions to access the list part of a string. There is also a cost associated with calculating the new number of seen characters during a string tail operation. When `snew` is applied to a list l, it returns l paired with 0 with an associated cost of one application. Because the abbreviation `sseen` is not an abstraction, it does not have the cost of an application, just the cost of obtaining the second element of the pair. Figure 5.2 contains these properties, which we use to calculate the meanings of the subprograms. Remember that the notation n_k represents the morphism from $\mathbf{1}$ to $\mathbf{N}$ representing the natural number k.

For the main program, it will be useful to define a few abbreviations, particularly for the value part of a continuation. First, let

$$
\begin{array}{rcl}
\mathbf{cont} & = & \mathbf{string} \to \mathbf{bool} \\
\mathbf{acctype} & = & \mathbf{regexp} \to \mathbf{cont} \to \mathbf{string} \to \mathbf{bool}
\end{array}
$$

The type **cont** represents continuations, and **acctype** represents the type of `acc`.

Next, we would prefer to use the regular expression notation from the beginning of this section instead of the morphisms themselves. For any object X in $\mathbf{PDom}$, if (h, d) is a morphism from $\mathbf{1}$ to id_X, then the following diagram must commute:

$$
\begin{tikzcd}
\mathbf{1} \arrow[r, "\mathsf{id}"] \arrow[d, "h"'] & \mathbf{1} \arrow[d, "d"] \\
X \arrow[r, "\mathsf{id}"'] & X
\end{tikzcd}
$$

Therefore $h = d$. What this means is that any element in the set X is uniquely identified with a morphism from $\mathbf{1}$ to id_X in $\mathbf{PDom}^{\to}$. Therefore we can identify the set of regular expressions with the elements of R^I. To do this we cannot assume that sequencing and choice are associative; for instance, $(a|b)|c$ is equivalent to the morphism $\mathsf{ror} \circ \langle \mathsf{ror} \circ \langle \mathsf{rchar} \circ a, \mathsf{rchar} \circ b\rangle, \mathsf{rchar} \circ c\rangle$, but $a|(b|c)$ is equivalent to the morphism $\mathsf{ror} \circ \langle \mathsf{rchar} \circ a, \mathsf{ror} \circ \langle \mathsf{rchar} \circ b, \mathsf{rchar} \circ c\rangle\rangle$, and there is no particular reason to believe that these two morphisms are equal. In particular, in examples 6 and 7 we show that there may be different costs associated with applying `accept` to these two regular expressions.

Lastly, there are three types of continuations possible: the initial continuation, one built from `acc`, and one built from `strcheck`. We will frequently want to represent the value portions (the portions without external cost) of these continuations. The initial continuation, `snil?`, is a value and thus factors through η. Therefore there exists a morphism $\mathsf{initk} : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$ such that

$$\eta \circ \mathsf{initk} = \mathcal{V}[\![\mathtt{snil?} : \mathbf{string} \to \mathbf{bool}]\!]$$

For `strcheck`, given morphisms $k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$, $r : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{regexp}]\!]$, and $s : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!]$,

$$\begin{array}{l}\mathcal{V}[\![\mathsf{a} : \mathbf{acctype}, \Gamma_1 \vdash \mathtt{strcheck} : \mathbf{cont}]\!] \circ \langle\rangle(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ k, \eta \circ s, \eta \circ r) \\ \quad = \mathcal{V}[\![\Gamma_1 \vdash [\mathsf{acc/a}]\mathtt{strcheck} : \mathbf{cont}]\!] \circ \langle\rangle(\eta \circ k, \eta \circ s, \eta \circ r) \\ \quad = \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma_1, \mathsf{s}' : \mathbf{string} \vdash e : \mathbf{bool}]\!] \circ (\langle\rangle(\eta \circ k, \eta \circ s, \eta \circ r) \times \eta))\end{array}$$

where

```
e = if (sseen(s) = sseen(s$'$)) then false
         else (acc (Star r1) k s$'$)
```

that is, $[\mathsf{acc/a}]\mathtt{strcheck} = \mathtt{lam\ s}'.e$, and

$$\Gamma_1 = \mathtt{k} : \mathbf{cont}, \mathtt{s} : \mathbf{string}, \mathtt{r1} : \mathbf{regexp}$$

Thus given k, r, and s as above, there exists a morphism $\mathsf{checkk}(k, s, r)$ from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\mathbf{string} \to \mathbf{bool}]\!]$ defined as

$$\mathsf{checkk}(k, s, r) = \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma_1, \mathsf{s}' : \mathbf{string} \vdash e : \mathbf{bool}]\!] \circ (\langle\rangle(\eta \circ k, \eta \circ s, \eta \circ r) \times \eta))$$

and such that

$$\eta \circ \mathsf{checkk}(k, s, r) = \mathcal{V}[\![\mathsf{a} : \mathbf{acctype}, \Gamma_1 \vdash \mathtt{strcheck} : \mathbf{cont}]\!] \circ \langle\rangle(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ k, \eta \circ s, \eta \circ r)$$

The morphism $\mathsf{checkk}(k, s, r)$ is the value portion of the meaning of `strcheck` with k, s, and r denoting the values of the three free variables `k`, `s`, and `r1`.

Lastly, for `acc`, given $r : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{regexp}]\!]$, $k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$, and the type assignment $\Gamma_2 = \mathsf{a} : \mathbf{acctype}, \mathtt{r} : \mathbf{regexp}, \mathtt{k} : \mathbf{cont}$,

$$\begin{array}{l}\mathbf{apply}(\mathcal{V}[\![\mathsf{acc}]\!], r, k) \\ \quad = [\![2t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathcal{V}[\![\Gamma_2 \vdash \mathtt{lam\ s}.e' : \mathbf{cont}]\!] \circ \langle\rangle(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ r, \eta \circ k) \\ \quad = [\![2t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma_2, \mathtt{s} : \mathbf{string} \vdash e' : \mathbf{bool}]\!] \circ (\mathsf{id} \times \eta)) \\ \qquad \circ \langle\rangle(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ r, \eta \circ k)\end{array}$$

where

```
e' = rcase r of CHAR: accchar OR: accor SEQ: accseq STAR: accstar
```

Thus for any r, k as above there exists a morphism $\mathsf{acck}(r, k)$ from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$ defined as

$$\mathsf{acck}(r, k) = \mathsf{curry}([\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma_2, \mathtt{s} : \mathbf{string} \vdash e' : \mathbf{bool}]\!] \circ (\mathsf{id} \times \eta)) \circ \langle\rangle(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ r, \eta \circ k)$$

and such that

$$\mathbf{apply}(\mathcal{V}[\![\mathsf{acc}]\!], r, k) = [\![2t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathsf{acck}(r, k)$$

The morphism $\mathsf{acck}(r,k)$ is the value part of the meaning of `acc` partially applied to r and k; it does not include the costs associated with unrolling the recursion or applying the two variables. It also lacks the external cost structure. It is the value portion of a continuation formed from `acc`, r, and k.

We also form an abbreviation for the fully applied `acc` function. Let

$$\mathsf{acc}(r,k,s) = \mathbf{apply}(\eta \circ \mathsf{acck}(r,k), s)$$

Then

$$\mathbf{apply}(\mathcal{V}[\![\mathsf{acc}]\!], r, k, s) = [\![2t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathsf{acc}(r,k,s)$$

Because $\mathsf{acc}(r,k,s)$ is derived from the continuation it also lacks the cost associated with applying r and k and unrolling the recursion once. Otherwise it is the same as applying the recursive function to its three arguments. We use $\mathsf{acc}(r,k,s)$ so we can uniformly handle sub-calls to `acc` both when they occur through recursion and when they occur through a continuation.

We now look at the subprograms. For clarity, we assign a function for each subprogram which supplies values for the free variables. Therefore, for $k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$, $s : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!]$, and $\Gamma = \mathtt{a} : \mathbf{acctype}, \mathtt{k} : \mathbf{cont}, \mathtt{s} : \mathbf{string}$, let

$$\begin{aligned}
\mathsf{accchar}(k,s) &= \mathcal{V}[\![\Gamma \vdash \mathtt{accchar} : \mathbf{nat} \to \mathbf{bool}]\!] \circ \langle\rangle(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ k, \eta \circ s) \\
\mathsf{accor}(k,s) &= \mathcal{V}[\![\Gamma \vdash \mathtt{accor} : \mathbf{nat} \to \mathbf{bool}]\!] \circ \langle\rangle(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ k, \eta \circ s) \\
\mathsf{accseq}(k,s) &= \mathcal{V}[\![\Gamma \vdash \mathtt{accseq} : \mathbf{nat} \to \mathbf{bool}]\!] \circ \langle\rangle(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ k, \eta \circ s) \\
\mathsf{accstar}(k,s) &= \mathcal{V}[\![\Gamma \vdash \mathtt{accstar} : \mathbf{nat} \to \mathbf{bool}]\!] \circ \langle\rangle(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ k, \eta \circ s)
\end{aligned}$$

These are the meanings of `accchar`, `accor`, `accseq`, and `accstar` where the free variable `a` (which in the program referred to the recursive function) is set to the meaning of `acc`.

We examine the simplest case first: sequencing. Let r_1, r_2 be regular expressions, i.e, morphisms from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\mathbf{regexp}]\!]$, let k be a continuation, i.e., a morphism from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$, and s be a string, i.e., a morphism from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!]$. Then, for $\Gamma_3 = \Gamma, \mathtt{r1} : \mathbf{regexp}, \mathtt{r2} : \mathbf{regexp}$,

$$\begin{aligned}
&\mathbf{apply}(\mathsf{accseq}(k,s), r_1, r_2) \\
&\quad= [\![2t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma_3 \vdash \mathtt{a\ r1\ (a\ r2\ k)} : \mathbf{bool}]\!] \circ \langle\rangle(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ k, \eta \circ s, \eta \circ r_1, \eta \circ r_2) \\
&\quad= [\![2t_{\mathrm{app}}]\!] \circ \mathbf{capply}(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ r_1, \mathbf{capply}(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ r_2, \eta \circ k), \eta \circ s) \\
&\quad= [\![2t_{\mathrm{app}}]\!] \circ \mathbf{capply}(\mathcal{V}[\![\mathsf{acc}]\!], \eta \circ r_1, [\![2t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{acck}(r_2,k), \eta \circ s) \\
&\quad= [\![4t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathbf{apply}(\mathcal{V}[\![\mathsf{acc}]\!], r_1, \mathsf{acck}(r_2,k), s) \\
&\quad= [\![6t_{\mathrm{app}} + 2t_{\mathrm{rec}}]\!] \circ \mathsf{acc}(r_1, \mathsf{acck}(r_2,k), s)
\end{aligned}$$

This means that $\mathsf{accseq}(k,s)$ applies `acc` to r_1, a continuation built from r_2 and k, and the s. In the process there were six applications and two recursion unrolls corresponding to the application of `accseq` and two applications of `acc`.

The `Or` and `Star` cases are only slightly more complicated:

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathbf{apply}(\mathsf{accor}(k,s), r_1, r_2)} \\
= & [\![2t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma_3 \vdash \texttt{(a r1 k s) orelse (a r2 k s)} : \mathbf{bool}]\!] \\
 & \quad \circ \langle\rangle(\mathcal{V}[\![\texttt{acc}]\!], \eta \circ k, \eta \circ s, \eta \circ r_1, \eta \circ r_2) \\
= & [\![2t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma_3 \vdash \texttt{if (a r1 k s) then true else (a r2 k s)} : \mathbf{bool}]\!] \\
 & \quad \circ \langle\rangle(\mathcal{V}[\![\texttt{acc}]\!], \eta \circ k, \eta \circ s, \eta \circ r_1, \eta \circ r_2) \\
= & [\![2t_{\mathrm{app}}]\!] \circ \ \mathbf{cond}(\mathbf{apply}(\mathcal{V}[\![\texttt{acc}]\!], r_1, k, s)) \ \mathbf{of} \\
 & \qquad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
 & \qquad \mathbf{False:} \quad [\![t_{\mathrm{false}}]\!] \circ \mathbf{apply}(\mathcal{V}[\![\texttt{acc}]\!], r_2, k, s) \\
= & [\![4t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \ \mathbf{cond}(\mathsf{acc}(r_1, k, s)) \ \mathbf{of} \\
 & \qquad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
 & \qquad \mathbf{False:} \quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ \mathsf{acc}(r_2, k, s)
\end{array}$$

Thus $\mathsf{accor}(k,s)$ applies acc to r_1, k, and s. If the result is true, then $\mathsf{accor}(k,s)$ is tt with cost of $4t_{\mathrm{app}}+t_{\mathrm{rec}}+t_{\mathrm{true}}$ plus the cost of evaluating $\mathsf{acc}(r_1,k,s)$. If the result is false, however, accor is then the value of $\mathsf{acc}(r_2,k,s)$ with the additional cost of evaluating $\mathsf{acc}(r_1,k,s)$ plus $6t_{\mathrm{app}}+2t_{\mathrm{rec}}+t_{\mathrm{false}}$.

Similarly,

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathbf{apply}(\mathsf{accstar}(k,s), r_1)} \\
= & [\![2t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma_0, \texttt{r1} : \mathbf{regexp} \vdash \texttt{(k s) orelse (a r1 strcheck s)} : \mathbf{bool}]\!] \\
 & \quad \circ \langle\rangle(\mathcal{V}[\![\texttt{acc}]\!], \eta \circ k, \eta \circ s, \eta \circ r_1) \\
= & [\![2t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma_0, \texttt{r1} : \mathbf{regexp} \vdash \texttt{if (k s) then true else (a r1 strcheck s)} : \mathbf{bool}]\!] \\
 & \quad \circ \langle\rangle(\mathcal{V}[\![\texttt{acc}]\!], \eta \circ k, \eta \circ s, \eta \circ r_1) \\
= & [\![2t_{\mathrm{app}}]\!] \circ \ \mathbf{cond}(\mathbf{apply}(\eta \circ k, s)) \ \mathbf{of} \\
 & \qquad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
 & \qquad \mathbf{False:} \quad [\![t_{\mathrm{false}}]\!] \circ \mathbf{apply}(\mathcal{V}[\![\texttt{acc}]\!], r_1, \mathsf{checkk}(k, s, r_1), s) \\
= & [\![2t_{\mathrm{app}}]\!] \circ \ \mathbf{cond}(\mathbf{apply}(\eta \circ k, s)) \ \mathbf{of} \\
 & \qquad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
 & \qquad \mathbf{False:} \quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ \mathsf{acc}(r_1, \mathsf{checkk}(k, s, r_1), s)
\end{array}$$

In this case the $\mathsf{accstar}(k,s)$ first applies the continuation k to s. If the result is true, then the result of accstar is true, with a cost of $2t_{\mathrm{app}}+t_{\mathrm{true}}$ plus the cost of applying k to s. If the result is false, then the result of accstar is the result of applying acc to r_1, a continuation made with checkk, and s, with the additional cost of $4t_{\mathrm{app}}+t_{\mathrm{rec}}+t_{\mathrm{false}}$ plus the cost of applying k to s.

For `accchar` we can better see its meaning if we compute it with different inputs. First, for any integer i (denoting an arbitrary "length" counter), let $\mathsf{snil}(i) = \langle \mathsf{nil}^I, \mathsf{n}_i \rangle$, and for any string $s = \langle [a_1, \ldots, a_n], \mathsf{n}_i \rangle$ and character a, let $\mathsf{scons}(a,s) = \langle [a, a_1, \ldots, a_n], \mathsf{n}_{i-1} \rangle$. Then, for any integer i and character element $a : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{nat}]\!]$,

$$\mathbf{apply}(\mathsf{accchar}(k, \mathsf{snil}(i)), a) = \quad [\![t_{\mathrm{true}} + 2t_{\mathrm{app}} + t_{\mathrm{fst}} + t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathsf{ff}$$

Thus applying accchar to an empty string results in a value of false (correctly indicating that there is not a match), with a cost of

$$t_{\mathrm{true}} + 2t_{\mathrm{app}} + t_{\mathrm{fst}} + t_{\mathrm{nil?}}$$

Of this cost the value $t_{\mathrm{app}}+t_{\mathrm{fst}}+t_{\mathrm{nil?}}$ is from the test snil (which is an abbreviation for `lams.nil?(fsts)`), the other t_{app} comes from applying the string and the cost t_{true} is because the program takes the first branch when the string does prove to be empty.

For the non-empty case, let s be any string. Then, for any characters $a, a' : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{nat}]\!]$,

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathcal{V}[\![\mathtt{s} : \mathbf{string}, \mathtt{n} : \mathbf{nat} \vdash \mathtt{shead(s) = n} : \mathbf{bool}]\!] \circ \langle\rangle(\eta \circ \mathsf{scons}(a', s), a)} \\
= & \mathbf{capply}(\mathcal{V}[\![=]\!], [\![t_{\text{app}} + t_{\text{fst}} + t_{\text{head}}]\!] \circ \eta \circ a', \eta \circ a) \\
= & [\![t_{\text{app}} + t_{\text{fst}} + t_{\text{head}}]\!] \circ \mathbf{apply}(\mathcal{V}[\![=]\!], a', a) \\
= & [\![t_{\text{app}} + t_{\text{fst}} + t_{\text{head}}]\!] \circ \mathsf{cond} \circ \langle \mathbf{bool}(a' = a), \\
 & \quad \langle [\![t_=]\!] \circ \eta \circ \mathbf{bool}(a' = a), [\![t_{\neq}]\!] \circ \eta \circ \mathbf{bool}(a' = a)\rangle\rangle \\
= & [\![t_{\text{app}} + t_{\text{fst}} + t_{\text{head}}]\!] \circ \mathbf{cond}(\{a' = a\}, [\![t_=]\!] \circ \eta \circ \mathsf{tt}, [\![t_{\neq}]\!] \circ \eta \circ \mathsf{ff})
\end{array}$$

The cost here is from applying the function `shead`. Therefore, for $\Gamma_4 = \mathtt{k} : \mathbf{acctype}, \mathtt{s} : \mathbf{string}, \mathtt{n} : \mathbf{nat}$

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathbf{apply}(\mathsf{accchar}(k, \mathsf{scons}(c', s)), c)} \\
= & [\![t_{\text{app}}]\!] \circ \mathcal{V}[\![\Gamma_4 \vdash \mathtt{(shead(s) = n)\ andalso\ (k(stail(s)))} : \mathbf{bool}]\!] \\
 & \quad \circ \langle\rangle(\eta \circ k, \eta \circ \mathsf{scons}(c', s), \eta \circ c) \\
= & [\![t_{\text{app}}]\!] \circ \mathcal{V}[\![\Gamma_4 \vdash \mathtt{if\ shead(s) = n\ then\ k(stail(s))\ else\ false} : \mathbf{bool}]\!] \\
 & \quad \circ \langle\rangle(\eta \circ k, \eta \circ \mathsf{scons}(c', s), \eta \circ c) \\
= & [\![t_{\text{false}} + 2t_{\text{app}} + t_{\text{fst}} + t_{\text{nil?}}]\!] \\
 & \quad \circ\ \mathbf{cond}([\![t_{\text{app}} + t_{\text{fst}} + t_{\text{head}}]\!] \circ \mathbf{cond}(\{c' = c\}, [\![t_=]\!] \circ \eta \circ \mathsf{tt}, [\![t_{\neq}]\!] \circ \eta \circ \mathsf{ff}))\ \mathbf{of} \\
 & \qquad \mathbf{True:}\quad [\![t_{\text{true}}]\!] \circ \mathbf{capply}(\eta \circ k, \mathbf{apply}(\mathcal{V}[\![\mathtt{stail}]\!], \mathsf{scons}(c', s))) \\
 & \qquad \mathbf{False:}\quad [\![t_{\text{false}}]\!] \circ \eta \circ \mathsf{ff} \\
= & [\![t_{\text{false}} + 3t_{\text{app}} + 2t_{\text{fst}} + t_{\text{nil?}} + t_{\text{head}}]\!] \\
 & \quad \circ\ \mathbf{cond}(\{c' = c\})\ \mathbf{of} \\
 & \qquad \mathbf{True:}\quad [\![t_{\text{true}} + t_=]\!] \circ \mathbf{capply}(\eta \circ k, [\![t_{\text{app}} + t_{\text{fst}} + t_+ + t_{\text{snd}} + t_{\text{tail}}]\!] \circ \eta \circ s) \\
 & \qquad \mathbf{False:}\quad [\![t_{\text{false}} + t_{\neq}]\!] \circ \eta \circ \mathsf{ff} \\
= & [\![t_{\text{false}} + 3t_{\text{app}} + 2t_{\text{fst}} + t_{\text{nil?}} + t_{\text{head}}]\!] \\
 & \quad \circ\ \mathbf{cond}(\{c' = c\})\ \mathbf{of} \\
 & \qquad \mathbf{True:}\quad [\![t_{\text{app}} + t_{\text{fst}} + t_+ + t_{\text{snd}} + t_{\text{tail}} + t_{\text{true}} + t_=]\!] \circ \mathbf{apply}(\eta \circ k, s) \\
 & \qquad \mathbf{False:}\quad [\![t_{\text{false}} + t_{\neq}]\!] \circ \eta \circ \mathsf{ff}
\end{array}$$

This means that if $\mathsf{accchar}$ is applied to a non-empty string s, the first character of s is compared to the character c. If that match succeeds, then the continuation k is applied to the rest of s and that result is returned with an additional cost of

$$4t_{\text{app}} + 3t_{\text{fst}} + t_{\text{snd}} + t_{\text{nil?}} + t_{\text{head}} + t_{\text{tail}} + t_{\text{false}} + t_{\text{true}} + t_= + t_+$$

If the match fails, then the result is false, with a cost of

$$3t_{\text{app}} + 2t_{\text{fst}} + t_{\text{nil?}} + t_{\text{head}} + 2t_{\text{false}} + t_{\neq}$$

Next we need to find the meaning of `strcheck`. First note that, for suffix strings $\langle l_1, \mathsf{n}_i\rangle$ and $\langle l_2, \mathsf{n}_j\rangle$ of s,

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathcal{V}[\![\mathtt{s} : \mathbf{string}, \mathtt{s'} : \mathbf{string} \vdash \mathtt{sseen(s) = sseen(s')} : \mathbf{bool}]\!] \circ \langle\rangle(\eta \circ \langle l_1, \mathsf{n}_i\rangle, \eta \circ \langle l_2, \mathsf{n}_j\rangle)} \\
= & [\![2t_{\text{snd}}]\!] \circ \mathbf{cond}(\{i = j\}, [\![t_=]\!] \circ \eta \circ \mathsf{tt}, [\![t_{\neq}]\!] \circ \eta \circ \mathsf{ff})
\end{array}$$

Therefore

$$\begin{array}{lll}
\mathbf{apply}(\eta \circ \mathsf{checkk}(k, \langle l, \mathsf{n}_i \rangle, r), \langle l', \mathsf{n}_j \rangle) & & \\
\quad = & [\![t_{\mathrm{app}}]\!] \circ & \mathbf{cond}([\![2t_{\mathrm{snd}}]\!] \circ \mathbf{cond}(\{i = j\}, [\![t_{=}]\!] \circ \eta \circ \mathsf{tt}, [\![t_{\neq}]\!] \circ \eta \circ \mathsf{ff}))\ \mathbf{of} \\
 & & \quad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{ff} \\
 & & \quad \mathbf{False:} \quad [\![t_{\mathrm{false}}]\!] \circ \eta \circ \mathbf{apply}(\mathcal{V}[\![\mathtt{acc}]\!], \mathsf{rstar} \circ r, k, \langle l', \mathsf{n}_j \rangle) \\
\quad = & [\![2t_{\mathrm{snd}} + t_{\mathrm{app}}]\!] \circ & \mathbf{cond}(\{i = j\})\ \mathbf{of} \\
 & & \quad \mathbf{True:} \quad [\![t_{\mathrm{true}} + t_{=}]\!] \circ \eta \circ \mathsf{ff} \\
 & & \quad \mathbf{False:} \quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}} + t_{\neq}]\!] \circ \mathsf{acc}(r^*, k, \langle l', \mathsf{n}_j \rangle)
\end{array}$$

This means that if the input string has the same length as the string in the `strcheck` function, the result of applying `strcheck` is a value of false with a cost of $2t_{\mathrm{snd}} + t_{\mathrm{app}} + t_{\mathrm{true}} + t_{=}$. If the string has a different length, however, the result is obtained by applying `acc` to that string with the original continuation and an additional cost of $2t_{\mathrm{snd}} + 3t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}} + t_{\neq}$.

For the main programs, we have that, for $r : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{regexp}]\!]$ and $l : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{list}(\mathbf{nat})]\!]$:

$$\begin{array}{ll}
\mathbf{apply}(\mathcal{V}[\![\mathtt{accept}]\!], r, l) & \\
\quad = & \mathbf{capply}(\mathcal{V}[\![\mathtt{acc}]\!], \eta \circ r, \mathcal{V}[\![\mathtt{snil?}]\!], \mathbf{apply}(\mathcal{V}[\![\mathtt{snew}]\!], l)) \\
\quad = & [\![t_{\mathrm{app}}]\!] \circ \mathbf{apply}(\mathcal{V}[\![\mathtt{acc}]\!], r, \mathsf{initk}, \langle l, \mathsf{n}_0 \rangle) \\
\quad = & [\![3t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathsf{acc}(r, \mathsf{initk}, \langle l, \mathsf{n}_0 \rangle)
\end{array}$$

For `acc` it is cleaner if we treat each type of regular expression separately. Therefore let $k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$, $a : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{nat}]\!]$, and $\Gamma_4 : \mathtt{a} : \mathbf{acctype}, \mathtt{r} : \mathbf{regexp}, \mathtt{k} : \mathbf{cont}, \mathtt{s} : \mathbf{string}$. Then, for any string $s : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!]$,

$$\begin{array}{rcl}
\mathsf{acc}(a, k, s) & = & [\![t_{\mathrm{app}}]\!] \circ \mathcal{V}[\![\Gamma_4 \vdash \mathtt{rcase}\ \ldots : \mathbf{bool}]\!] \circ \langle\rangle(\mathcal{V}[\![\mathtt{acc}]\!], \mathsf{rchar} \circ a, k, s) \\
 & = & [\![t_{\mathrm{rcase}} + t_{\mathrm{app}}]\!] \circ \mathsf{app1} \circ \langle a, \mathsf{vaccchar}(k, s) \rangle \\
 & = & [\![t_{\mathrm{rcase}} + t_{\mathrm{app}}]\!] \circ \mathsf{app} \circ \langle \mathsf{vaccchar}(k, s), a \rangle \\
 & = & [\![t_{\mathrm{rcase}} + t_{\mathrm{app}}]\!] \circ \mathbf{apply}(\mathsf{accchar}(k, s), a)
\end{array}$$

where $\mathsf{vaccchar}(k, s)$ is the value part of $\mathsf{accchar}(k, s)$, i.e., $\mathsf{accchar}(k, s) = \eta \circ \mathsf{vaccchar}(k, s)$.

Similarly, for $r, r' : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{regexp}]\!]$,

$$\begin{array}{rcl}
\mathsf{acc}(r^*, k, s) & = & [\![t_{\mathrm{rcase}} + t_{\mathrm{app}}]\!] \circ \mathbf{apply}(\mathsf{accstar}(k, s), r) \\
\mathsf{acc}(r_1 | r_2, k, s) & = & [\![t_{\mathrm{rcase}} + t_{\mathrm{app}}]\!] \circ \mathbf{apply}(\mathsf{accor}(k, s), r_1, r_2) \\
\mathsf{acc}(r_1 r_2, k, s) & = & [\![t_{\mathrm{rcase}} + t_{\mathrm{app}}]\!] \circ \mathbf{apply}(\mathsf{accseq}(k, s), r_1, r_2)
\end{array}$$

Thus for each type of expression, the meaning of `acc` depends on the meaning of one of the four subprograms defined earlier. The semantics of the `accept` program are summarized as follows:

Theorem 5.1.3 *The following equations hold for any regular expression $r : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{regexp}]\!]$, list* *of integers* $l : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{list}(\mathbf{nat})]\!]$*, string* $s : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!]$*, and continuation* $k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$*:*

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathbf{apply}(\mathcal{V}[\![\mathtt{accept}]\!], r, l)} \\
 & = & [\![3t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \mathsf{acc}(r, \mathsf{initk}, \langle l, \mathsf{n}_0\rangle) \\
\mathsf{acc}(a, k, s) & = & [\![t_{\mathrm{rcase}} + t_{\mathrm{app}}]\!] \circ \mathbf{apply}(\mathsf{accchar}(k, s), a) \\
\mathsf{acc}(r^*, k, s) & = & [\![t_{\mathrm{rcase}} + t_{\mathrm{app}}]\!] \circ \mathbf{apply}(\mathsf{accstar}(k, s), r) \\
\mathsf{acc}(r_1|r_2, k, s) & = & [\![t_{\mathrm{rcase}} + t_{\mathrm{app}}]\!] \circ \mathbf{apply}(\mathsf{accor}(k, s), r_1, r_2) \\
\mathsf{acc}(r_1 r_2, k, s) & = & [\![t_{\mathrm{rcase}} + t_{\mathrm{app}}]\!] \circ \mathbf{apply}(\mathsf{accseq}(k, s), r_1, r_2) \\
\multicolumn{3}{l}{\mathbf{apply}(\mathsf{accchar}(k, \mathsf{snil}(i)), a)} \\
 & = & [\![t_{\mathrm{true}} + 2t_{\mathrm{app}} + t_{\mathrm{fst}} + t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathsf{ff} \\
\multicolumn{3}{l}{\mathbf{apply}(\mathsf{accchar}(k, \mathsf{scons}(c', s)), c)} \\
 & = & [\![t_{\mathrm{false}} + 3t_{\mathrm{app}} + 2t_{\mathrm{fst}} + t_{\mathrm{nil?}} + t_{\mathrm{head}}]\!] \\
 & & \quad \circ\ \mathbf{cond}(\{c' = c\})\ \mathbf{of} \\
 & & \qquad \mathbf{True:}\quad [\![t_{\mathrm{app}} + t_{\mathrm{fst}} + t_{+} + t_{\mathrm{snd}} + t_{\mathrm{tail}} + t_{\mathrm{true}} + t_{=}]\!] \circ \mathbf{apply}(\eta \circ k, s) \\
 & & \qquad \mathbf{False:}\quad [\![t_{\mathrm{false}} + t_{\neq}]\!] \circ \eta \circ \mathsf{ff} \\
\multicolumn{3}{l}{\mathbf{apply}(\mathsf{accstar}(k, s), r)} \\
 & = & [\![2t_{\mathrm{app}}]\!] \circ\ \mathbf{cond}(\mathbf{apply}(\eta \circ k, s))\ \mathbf{of} \\
 & & \qquad \mathbf{True:}\quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
 & & \qquad \mathbf{False:}\quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ \mathsf{acc}(r, \mathsf{checkk}(k, s, r), s) \\
\multicolumn{3}{l}{\mathbf{apply}(\mathsf{accor}(k, s), r_1, r_2)} \\
 & = & [\![4t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ\ \mathbf{cond}(\mathsf{acc}(r_1, k, s))\ \mathbf{of} \\
 & & \qquad \mathbf{True:}\quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
 & & \qquad \mathbf{False:}\quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ \mathsf{acc}(r_2, k, s) \\
\multicolumn{3}{l}{\mathbf{apply}(\mathsf{accseq}(k, s), r_1, r_2)} \\
 & = & [\![6t_{\mathrm{app}} + 2t_{\mathrm{rec}}]\!] \circ \mathsf{acc}(r_1, \mathsf{acck}(r_2, k), s) \\
\multicolumn{3}{l}{\mathbf{apply}(\eta \circ \mathsf{acck}(r, k), s)} \\
 & = & \mathsf{acc}(r, k, s) \\
\multicolumn{3}{l}{\mathbf{apply}(\eta \circ \mathsf{checkk}(k, \langle l, \mathsf{n}_i\rangle, r), \langle l', \mathsf{n}_j\rangle)} \\
 & = & [\![2t_{\mathrm{snd}} + t_{\mathrm{app}}]\!] \circ\ \mathbf{cond}(\{i = j\})\ \mathbf{of} \\
 & & \qquad \mathbf{True:}\quad [\![t_{\mathrm{true}} + t_{=}]\!] \circ \eta \circ \mathsf{ff} \\
 & & \qquad \mathbf{False:}\quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}} + t_{\neq}]\!] \circ \mathsf{acc}(r^*, k, \langle l', \mathsf{n}_j\rangle) \\
\multicolumn{3}{l}{\mathbf{apply}(\eta \circ \mathsf{initk}, \langle [a_1, \ldots, a_n], \mathsf{n}_k\rangle)} \\
 & = & [\![t_{\mathrm{app}} + t_{\mathrm{fst}} + t_{\mathrm{nil?}}]\!] \circ \eta \circ \mathbf{bool}(n = 0)
\end{array}
$$

5.1.5 Examples

Before discussing general properties involving complexity, we shall look at some concrete examples, that show how varied complexity can be for different types of regular expressions. For these examples, let a^n be the list of n a's.

To make the calculations easier, we would prefer not to concern ourselves with non-terminating results (i.e., values of $\perp$). Fortunately, it is relatively straightforward to show that `accept` terminates on "normal" inputs; more importantly, that its value (when fully applied) will always have the form $[\![t]\!] \circ \eta \circ b$ for some cost t and morphism $b : \mathbf{1} \to \mathbf{Bool}$.

Definition 5.1.1 A continuation $k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$ *terminates* if for all strings s from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!]$, there exist a cost t and a morphism b from $\mathbf{1}$ to $\mathbf{Bool}$ such that

$$\mathbf{apply}(\eta \circ k, s) = [\![t]\!] \circ \eta \circ b$$

Theorem 5.1.4 *For any regular expression $r : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{regexp}]\!]$, continuation $k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$ and string $s : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!]$, if k terminates, then* acc *terminates, i.e., there exist a cost t and a morphism $b : \mathbf{1} \to \mathbf{Bool}$ such that*

$$\mathsf{acc}(r, k, s) = [\![t]\!] \circ \eta \circ b$$

Proof. By induction on both the structure of r and the length of s. The proof actually shows a stronger result, namely that k only needs to terminate on s and its the suffixes. As part of the proof it also shows that if k terminates, so do acck and checkk. □

Example 1: $r = a_1|\ldots|a_n$, $s = \mathsf{scons}(a_i, s')$.

For this example n always refers to the number of expression choices, and i always refers to the choice that actually matches the prefix of the string in question. Assume that at least the first i of the characters a_j's are distinct. Suppose that k is a terminating continuation such that there exists a cost t such that $\mathbf{apply}(\eta \circ k, s') = [\![t]\!] \circ \eta \circ \mathsf{tt}$, i.e, k returns true (with cost t) on the tail of s.

As we are interested in the cost as i and n vary, and as k is assumed to terminate, we can assume that there exist functions $T_{\mathrm{or}}(i, n)$ and $V_{\mathrm{or}}(i, n)$ such that

$$\mathsf{acc}(r, k, s) = [\![T_{\mathrm{or}}(i, n)]\!] \circ \eta \circ V_{\mathrm{or}}(i, n)$$

We will now determine what these functions must be. This is done by examining various cases and building a collection of simple recurrence equations. The functions are then determined by solving the equations.

- Suppose that $n = 1$. Then r must be the single character a_1 and i must be 1, i.e., the first character of s is a_1 as well. Then

$$\begin{aligned}
&\mathsf{acc}(a_1, k, s)\\
&\quad = [\![5t_{\mathrm{app}} + t_{=} + t_{\mathrm{true}} + t_{\mathrm{false}} + 3t_{\mathrm{fst}} + t_{\mathrm{snd}} + t_{\mathrm{head}} + t_{\mathrm{tail}} + t_{\mathrm{nil?}} + t_{+} + t_{\mathrm{rcase}}]\!]\\
&\qquad\quad \circ \mathbf{apply}(\eta \circ k, s')\\
&\quad = [\![t + 5t_{\mathrm{app}} + t_{=} + t_{\mathrm{true}} + t_{\mathrm{false}} + 3t_{\mathrm{fst}} + t_{\mathrm{snd}} + t_{\mathrm{head}} + t_{\mathrm{tail}} + t_{\mathrm{nil?}} + t_{+} + t_{\mathrm{rcase}}]\!]\\
&\qquad\quad \circ \eta \circ \mathsf{tt}
\end{aligned}$$

  To simplify the rest of the calculations, let

$$t_0 = 5t_{\mathrm{app}} + t_{=} + t_{\mathrm{true}} + t_{\mathrm{false}} + 3t_{\mathrm{fst}} + t_{\mathrm{snd}} + t_{\mathrm{head}} + t_{\mathrm{tail}} + t_{\mathrm{nil?}} + t_{+} + t_{\mathrm{rcase}}$$

  We can therefore deduce that

$$\begin{aligned}
T_{\mathrm{or}}(1,1) &= t + t_0\\
V_{\mathrm{or}}(1,1) &= \mathsf{tt}
\end{aligned}$$

- Next suppose that $n > 1$ and $i = 1$, i.e., the first character of s is still a_1. Because we originally assumed that $(a_1|\ldots|a_n) = a_1|(a_2|\ldots|a_n)$,

$$\begin{aligned}
\mathsf{acc}(r,k,s) &= [\![5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}}]\!] \\
&\qquad \circ\ \textbf{cond}(\mathsf{acc}(a_1,k,s))\ \textbf{of} \\
&\qquad\quad \textbf{True:}\quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
&\qquad\quad \textbf{False:}\quad [\![2t_{\text{app}} + t_{\text{false}} + t_{\text{rec}}]\!] \circ \mathsf{acc}(a_2|\ldots|a_n,k,s) \\
&= [\![5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}}]\!] \circ [\![t_{\text{true}} + t + t_0]\!] \circ \eta \circ \mathsf{tt} \\
&= [\![t_{\text{true}} + t + t_0 + 5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}}]\!] \circ \eta \circ \mathsf{tt}
\end{aligned}$$

Thus we know that for $n > 1$,

$$\begin{aligned}
T_{\text{or}}(1,n) &= t_{\text{true}} + t + t_0 + 5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}} \\
V_{\text{or}}(1,n) &= \mathsf{tt}
\end{aligned}$$

Here is where the decision to make choice associate to the right makes the cost calculation simpler, as we do not also have calculate the cost to reach the first choice operator.

- Lastly suppose that $n > 1$ and $1 < i \leq n$. Because

$$\mathsf{acc}(a_1,k,s) = [\![4t_{\text{app}} + 2t_{\text{false}} + 2t_{\text{fst}} + t_{\text{head}} + t_{\neq} + t_{\text{nil?}} + t_{\text{rcase}}]\!] \circ \eta \circ \mathsf{ff}$$

we know that

$$\begin{aligned}
\mathsf{acc}(r,k,s) &= [\![5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}}]\!] \\
&\qquad \circ\ \textbf{cond}([\![t_0']\!] \circ \eta \circ \mathsf{ff})\ \textbf{of} \\
&\qquad\quad \textbf{True:}\quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
&\qquad\quad \textbf{False:}\quad [\![2t_{\text{app}} + t_{\text{false}} + t_{\text{rec}}]\!] \circ \mathsf{acc}(a_2|\ldots|a_n,k,s) \\
&= [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + t_{\text{false}} + t_0']\!] \circ \mathsf{acc}(a_2|\ldots|a_n,k,s) \\
&= [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + t_{\text{false}} + t_0' + T_{\text{or}}(i-1,n-1)]\!] \circ \eta \circ V_{\text{or}}(i-1,n-1)
\end{aligned}$$

where $t_0' = 4t_{\text{app}} + 2t_{\text{false}} + 2t_{\text{fst}} + t_{\text{head}} + t_{\neq} + t_{\text{nil?}} + t_{\text{rcase}}$.
From this we can deduce that

$$\begin{aligned}
T_{\text{or}}(i,n) &= 7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + t_{\text{false}} + t_0' + T(i-1,n-1) \\
V_{\text{or}}(i,n) &= V_{\text{or}}(i-1,n-1)
\end{aligned}$$

Thus it should be clear that the value $V_{\text{or}}(i,n)$ is always tt and that the cost is determined by the following equations:

$$\begin{aligned}
T_{\text{or}}(1,1) &= t + t_0 \\
T_{\text{or}}(1,n) &= t_{\text{true}} + t + t_0 + 5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}} \\
T_{\text{or}}(i,n) &= 7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + t_{\text{false}} + t_0' + T(i-1,n-1)
\end{aligned}$$

Solving these, and expanding t_0 and t_0' out again, we find that

$$\begin{aligned}
T_{\text{or}}(i,i) &= (11i-6)t_{\text{app}} + (3i-2)t_{\text{false}} + (2i+1)t_{\text{fst}} + i(t_{\text{head}} + t_{\text{nil?}}) + (i-1)t_{\neq} \\
&\quad + (2i-1)t_{\text{rcase}} + (2i-2)t_{\text{rec}} + t_{=} + t_{+} + t_{\text{snd}} + t_{\text{tail}} + t_{\text{true}} + t \\
T_{\text{or}}(i,n) &= T_{\text{or}}(i,i) + 5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}} + t_{\text{true}}
\end{aligned}$$

The difference in cost comes from expanding one additional choice expression when $i < n$. Intuitively, the cost $i(t_{\text{head}} + t_{\text{nil?}})$ refer to the number of times the program must compare a_i with the first character of s. In this case, the comparison fails $i-1$ times and succeeds once, thus there is the cost $(i-1)t_{\neq}$ plus a single $t_{=}$. Similarly, the factor $3n-2$ on t_{false} refers to the number of times the program checks that s is null, plus twice the number of times the first character of s fails to match the a_j $(1 \leq j < i-1)$, as each time that test fails both the single character test and the first part of a choice test fails with a cost of t_{false}. Overall the cost is linear in i.

Example 2: $r = a_1|\ldots|a_n$, $s = \mathsf{scons}(b, s')$.

This example is similar to Example 1, except that the first character of s is *not* any of the characters in r. Suppose that k is a terminating continuation. Then, as this example we are only varying the size n of r, we can assume that there exist functions T_{nomatch} and V_{nomatch} such that for any $n > 1$, $\mathsf{acc}(r, k, s) = [\![T_{\text{nomatch}}(n)]\!] \circ \eta \circ V_{\text{nomatch}}(n)$. We now determine what T_{nomatch} and V_{nomatch} must be by analyzing the various cases. If $n = 1$, then r is the single character a_1 and from the previous example we know that

$$\mathsf{acc}(r, k, s) = [\![t_0']\!] \circ \eta \circ \mathsf{ff}$$

where $t_0' = 4t_{\text{app}} + 2t_{\text{false}} + 2t_{\text{fst}} + t_{\text{head}} + t_{\neq} + t_{\text{nil?}} + t_{\text{rcase}}$.

From this we can deduce that

$$\begin{aligned} T_{\text{nomatch}}(1) &= t_0' \\ V_{\text{nomatch}}(1) &= \mathsf{ff} \end{aligned}$$

If $n > 1$ then from the previous example (in particular the case where $n > 1$ and $i > 1$) we know that

$$\begin{aligned} \mathsf{acc}(r, k, s) &= [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + t_{\text{false}} + t_0']\!] \circ \mathsf{acc}(a_2|\ldots|a_n, k, s) \\ &= [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + t_{\text{false}} + t_0']\!] \circ [\![T_{\text{nomatch}}(n-1)]\!] \circ \eta \circ V_{\text{nomatch}}(n-1) \end{aligned}$$

From this we know that the functions T_{nomatch} and V_{nomatch} must satisfy the following equations:

$$\begin{aligned} T_{\text{nomatch}}(1) &= t_0' \\ T_{\text{nomatch}}(n) &= T_{\text{nomatch}}(n-1) + 7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + t_{\text{false}} + t_0' \\ V_{\text{nomatch}}(1) &= \mathsf{ff} \\ V_{\text{nomatch}}(n) &= V_{\text{nomatch}}(n-1) \end{aligned}$$

Solving these equations we find that

$$\begin{aligned} T_{\text{nomatch}}(n) &= (n-1)(7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + t_{\text{false}}) + nt_0' \\ V_{\text{nomatch}}(n) &= \mathsf{ff} \end{aligned}$$

or, if we expand t_0',

$$\begin{aligned} T_{\text{nomatch}}(n) \;=\;& (11n-7)t_{\text{app}} + (3n-1)t_{\text{false}} + 2nt_{\text{fst}} + (2n-1)t_{\text{rcase}} + (2n-2)t_{\text{rec}} \\ &+ n(t_{\text{head}} + t_{\neq} + t_{\text{nil?}}) \end{aligned}$$

In this case it is easier to see where the cost arises. The costs in t_0' are all related to the test of a single character. The other costs are all related to managing the choice operator. As there are n characters and $n-1$ choices, the total cost is a simple combination of the two. Overall the cost is linear in the size of the regular expression.

Example 3: $r = a_1 \ldots a_n$, $s = \langle [a_1, \ldots, a_n], m \rangle$

For this example n refers to the number of characters in the sequence, and s is a string that exactly matches that sequence. Let k be any terminating continuation. Because it terminates we know that there exist a value $b : \mathbf{1} \to \mathbf{Bool}$ and a cost t_k such that

$$\mathbf{apply}(\eta \circ k, \mathsf{snil}) = [\![t_k]\!] \circ \eta \circ b$$

As we vary only the size n of the sequence, then we can also assume that there exist functions T_{seq} and V_{seq} such that

$$\mathsf{acc}(r, k, s) = [\![T_{\text{seq}}(n)]\!] \circ \eta \circ V_{\text{seq}}(n)$$

We can determine what T_{seq} and V_{seq} must be by looking at two cases for n:

- Suppose that $n = 1$. Then r is the single character test a_1 and s is a string consisting of the single character a_1. Therefore

$$\begin{aligned} \mathsf{acc}(r, k, s) = {} & \mathsf{acc}(a_1, k, s) \\ = {} & [\![t_0]\!] \circ \mathbf{apply}(\eta \circ k, \mathsf{snil}) \\ = {} & [\![t_k + t_0]\!] \circ \eta \circ b \end{aligned}$$

  where t_0 was defined on page 169 in Example 1. From this we can determine that

$$\begin{aligned} T_{\text{seq}}(1) = {} & t_k + t_0 \\ V_{\text{seq}}(1) = {} & b \end{aligned}$$

- Suppose that $n > 1$. Then r is a sequence and s has a length of more than 1. Therefore

$$\begin{aligned} \mathsf{acc}(r, k, s) = {} & [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}}]\!] \circ \mathsf{acc}(a_1, \mathsf{acck}([a_2 \ldots a_n], k), s) \\ = {} & [\![t_0 + 7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}}]\!] \circ \mathsf{acc}(a_2 \ldots a_n, k, \langle [a_2, \ldots, a_n], m+1 \rangle) \\ = {} & [\![T_{\text{seq}}(n-1) + t_0 + 7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}}]\!] \circ \eta \circ V_{\text{seq}}(n-1) \end{aligned}$$

  where acck is the continuation that applies acc for a regular expression consisting of the rest of the sequence. From this we know that the following equations must hold when $n > 1$:

$$\begin{aligned} T_{\text{seq}}(n) = {} & t_0 + 7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + T_{\text{seq}}(n-1) \\ V_{\text{seq}}(n) = {} & V_{\text{seq}}(n-1) \end{aligned}$$

We can thus determine the values of T_{seq} and V_{seq} by solving the following recurrence equations:

$$\begin{aligned} T_{\text{seq}}(1) = {} & t_k + t_0 \\ T_{\text{seq}}(n) = {} & t_0 + 7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + T_{\text{seq}}(n-1) \quad (n > 1) \\ V_{\text{seq}}(1) = {} & b \\ V_{\text{seq}}(n) = {} & V_{\text{seq}}(n-1) \quad (n > 1) \end{aligned}$$

Clearly for all $n \geq 1$, $V_{\text{seq}}(n)$ must be b, and, if we also expand t_0, we know that

$$\begin{aligned} T_{\text{seq}}(n) = {} & (12n-7)t_{\text{app}} + 3nt_{\text{fst}} + n(t_{=} + t_{\text{false}} + t_{\text{head}} + t_{\text{nil?}} + t_{+} + t_{\text{snd}} + t_{\text{tail}} + t_{\text{true}}) \\ & + (2n-1)t_{\text{rcase}} + (2n-2)t_{\text{rec}} + t_k \end{aligned}$$

Again the cost t_0 corresponds to successfully matching a single character, while the cost of $7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}}$ corresponds to expanding the sequence once. Therefore the total cost corresponds to n successful matches and $n-1$ sequence expansions. Thus overall, the cost is linear in the size of the regular expression which is the same as the length of the string.

Example 4: $r = a^*$, $s = a^n$.

In this example n (which can be 0) refers to the number of characters a in s. Let k be a terminating continuation that is false for all strings a^m when $m > 0$. Thus we know that for all $1 \leq m \leq n$, there exists a cost t_{k_m} such that

$$\mathbf{apply}(\eta \circ k, a^m) = [\![t_{k_m}]\!] \circ \eta \circ \mathsf{ff}$$

Because k terminates, there must also exist a cost t_{k_0} and a value $b_k : \mathbf{1} \to \mathbf{Bool}$ such that

$$\mathbf{apply}(\eta \circ k, \mathsf{snil}) = [\![t_{k_0}]\!] \circ \eta \circ b_k$$

An example of such a continuation is the initial continuation initk, where

$$t_{k_0} = t_{k_m} = t_{\text{app}} + t_{\text{fst}} + t_{\text{nil?}} \quad \text{and} \quad b_k = \mathsf{tt}$$

To see how the cost of applying acc as n changes and as b_k can vary, because k terminates we can assume that there exist functions T_{rep} and V_{rep} such that

$$\mathsf{acc}(r, k, s) = [\![T_{\text{rep}}(n, b_k)]\!] \circ \eta \circ V_{\text{rep}}(n, b_k)$$

We can determine the values of T_{rep} and V_{rep} by solving recurrence equations built from the cases where $n = 0$ and $b_k = \mathsf{tt}$, $n = 0$ and $b_k = \mathsf{ff}$, and $n > 0$.

- Suppose that $n = 0$ and $b_k = \mathsf{tt}$. Then $s = \mathsf{snil}$ so

$$\mathbf{apply}(\eta \circ k, s) = [\![t_{k_0}]\!] \circ \eta \circ b_k = [\![t_{k_0}]\!] \circ \eta \circ \mathsf{tt}$$

  Therefore

$$\begin{aligned}
\mathsf{acc}(r, k, s) &= [\![2t_{\text{app}} + t_{\text{rcase}}]\!] \\
&\qquad \circ\ \mathbf{cond}([\![t_{k_0}]\!] \circ \eta \circ \mathsf{tt})\ \mathbf{of} \\
&\qquad\qquad \mathbf{True:} \quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
&\qquad\qquad \mathbf{False:} \quad [\![2t_{\text{app}} + t_{\text{false}} + t_{\text{rec}}]\!] \circ \mathsf{acc}(a, \mathsf{checkk}(k, s, a), s) \\
&= [\![t_{\text{true}} + t_{k_0} + 2t_{\text{app}} + t_{\text{rcase}}]\!] \circ \eta \circ \mathsf{tt}
\end{aligned}$$

  From this we can deduce that

$$\begin{aligned}
T_{\text{rep}}(0, \mathsf{tt}) &= t_{\text{true}} + t_{k_0} + 2t_{\text{app}} + t_{\text{rcase}} \\
V_{\text{rep}}(0, \mathsf{tt}) &= \mathsf{tt}
\end{aligned}$$

- Suppose that $n = 0$ and $b_k = \mathsf{ff}$. Then s is again snil, so

$$\mathbf{apply}(\eta \circ k, s) = [\![t_{k_0}]\!] \circ \eta \circ b_k = [\![t_{k_0}]\!] \circ \eta \circ \mathsf{ff}$$

  Because s is snil, we know that

$$\mathsf{acc}(a, \mathsf{checkk}(k, s, a), s) = [\![t_{\text{rcase}} + 3t_{\text{app}} + t_{\text{true}} + t_{\text{fst}} + t_{\text{nil?}}]\!] \circ \eta \circ \mathsf{ff}$$

Then

$$\begin{aligned}
\mathsf{acc}(r,k,s) &= [\![2t_{\text{app}}+t_{\text{rcase}}]\!] \\
&\quad \circ\ \textbf{cond}([\![t_{k_0}]\!]\circ\eta\circ\mathsf{ff})\ \textbf{of} \\
&\qquad \textbf{True:}\quad [\![t_{\text{true}}]\!]\circ\eta\circ\mathsf{tt} \\
&\qquad \textbf{False:}\quad [\![5t_{\text{app}}+t_{\text{false}}+t_{\text{rec}}+t_{\text{rcase}}+t_{\text{true}}+t_{\text{fst}}+t_{\text{nil?}}]\!]\circ\eta\circ\mathsf{ff} \\
&= [\![t_{k_0}+2t_{\text{rcase}}+7t_{\text{app}}+t_{\text{false}}+t_{\text{rec}}+t_{\text{true}}+t_{\text{fst}}+t_{\text{nil?}}]\!]\circ\eta\circ\mathsf{ff}
\end{aligned}$$

From this we can derive the following:

$$\begin{aligned}
T_{\text{rep}}(0,\mathsf{ff}) &= t_{k_0}+2t_{\text{rcase}}+7t_{\text{app}}+t_{\text{false}}+t_{\text{rec}}+t_{\text{true}}+t_{\text{fst}}+t_{\text{nil?}} \\
V_{\text{rep}}(0,\mathsf{ff}) &= \mathsf{ff}
\end{aligned}$$

- Lastly, suppose that $n>0$. Then $\textbf{apply}(\eta\circ k,s)=[\![t_{k_n}]\!]\circ\eta\circ\mathsf{ff}$, so

$$\begin{aligned}
\mathsf{acc}(r,k,s) &= [\![2t_{\text{app}}+t_{\text{rcase}}]\!] \\
&\quad \circ\ \textbf{cond}([\![t_{k_n}]\!]\circ\eta\circ\mathsf{ff})\ \textbf{of} \\
&\qquad \textbf{True:}\quad [\![t_{\text{true}}]\!]\circ\eta\circ\mathsf{tt} \\
&\qquad \textbf{False:}\quad [\![2t_{\text{app}}+t_{\text{false}}+t_{\text{rec}}]\!]\circ\mathsf{acc}(a,\mathsf{checkk}(k,s,a),s) \\
&= [\![t_{k_n}+4t_{\text{app}}+t_{\text{false}}+t_{\text{rec}}+t_{\text{rcase}}]\!]\circ\mathsf{acc}(a,\mathsf{checkk}(k,s,a),s) \\
&= [\![t_0+t_{k_n}+4t_{\text{app}}+t_{\text{false}}+t_{\text{rec}}+t_{\text{rcase}}]\!] \\
&\quad \circ\,\textbf{apply}(\eta\circ\mathsf{checkk}(k,s,a),\langle a^{n-1},i+1\rangle) \\
&= [\![t_0+t_{k_n}+2t_{\text{snd}}+7t_{\text{app}}+2t_{\text{false}}+2t_{\text{rec}}+t_{\neq}]\!]\circ\mathsf{acc}(r,k,a^{n-1}) \\
&= [\![t_0+t_{k_n}+2t_{\text{snd}}+7t_{\text{app}}+2t_{\text{false}}+2t_{\text{rec}}+t_{\neq}+T_{\text{rep}}(n-1)]\!] \\
&\quad \circ\,\eta\circ V_{\text{rep}}(n-1)
\end{aligned}$$

  where t_0 is defined in Equation 5.1 in Example 1. From this we can derive the following equations:

$$\begin{aligned}
T_{\text{rep}}(n,b_k) &= t_0+t_{k_n}+2t_{\text{snd}}+7t_{\text{app}}+2t_{\text{false}}+2t_{\text{rec}}+t_{\neq}+T_{\text{rep}}(n-1) \\
V_{\text{rep}}(n,b_k) &= V_{\text{rep}}(n-1,b_k)
\end{aligned}$$

Thus we have the following set of equations:

$$\begin{aligned}
T_{\text{rep}}(0,\mathsf{tt}) &= t_{k_0}+t_{\text{true}}+2t_{\text{app}}+t_{\text{rcase}} \\
T_{\text{rep}}(0,\mathsf{ff}) &= t_{k_0}+2t_{\text{rcase}}+7t_{\text{app}}+t_{\text{false}}+t_{\text{rec}}+t_{\text{true}}+t_{\text{fst}}+t_{\text{nil?}} \\
T_{\text{rep}}(n,b_k) &= t_0+t_{k_n}+2t_{\text{snd}}+7t_{\text{app}}+2t_{\text{false}}+2t_{\text{rec}}+t_{\neq}+T_{\text{rep}}(n-1) \quad (n>0) \\
V_{\text{rep}}(0,\mathsf{tt}) &= \mathsf{tt} \\
V_{\text{rep}}(0,\mathsf{ff}) &= \mathsf{ff} \\
V_{\text{rep}}(n,b_k) &= V_{\text{rep}}(n-1,b_k) \quad (n>0)
\end{aligned}$$

By solving these equations and expanding t_0 we get that for all $n\geq 0$

$$\begin{aligned}
T_{\text{rep}}(n,\mathsf{tt}) &= n(3t_{\text{snd}}+12t_{\text{app}}+3t_{\text{false}}+2t_{\text{rec}}+t_{\neq}+t_{=}+t_{\text{true}}+3t_{\text{fst}}+t_{\text{head}}+t_{\text{tail}}) \\
&\qquad +n(t_{\text{nil?}}+t_{+}+t_{\text{rcase}})+t_{\text{true}}+2t_{\text{app}}+t_{\text{rcase}}+\textstyle\sum_{i=0}^{n}t_{k_i} \\
&= 3nt_{\text{snd}}+(12n+2)t_{\text{app}}+3nt_{\text{false}}+2nt_{\text{rec}}+nt_{\neq}+nt_{=}+(n+1)t_{\text{true}} \\
&\qquad +3nt_{\text{fst}}+nt_{\text{head}}+nt_{\text{tail}}+nt_{\text{nil?}}+nt_{+}+(n+1)t_{\text{rcase}}+\textstyle\sum_{i=0}^{n}t_{k_i} \\
T_{\text{rep}}(n,\mathsf{ff}) &= n(3t_{\text{snd}}+12t_{\text{app}}+3t_{\text{false}}+2t_{\text{rec}}+t_{\neq}+t_{=}+t_{\text{true}}+3t_{\text{fst}}+t_{\text{head}}+t_{\text{tail}}) \\
&\qquad +t_{\text{nil?}}+t_{+}+t_{\text{rcase}})+2t_{\text{rcase}}+7t_{\text{app}}+t_{\text{false}}+t_{\text{rec}}+t_{\text{true}}+t_{\text{fst}}+t_{\text{nil?}} \\
&\qquad +\textstyle\sum_{i=0}^{n}t_{k_i} \\
&= 3nt_{\text{snd}}+(12n+7)t_{\text{app}}+(3n+1)t_{\text{false}}+(2n+1)t_{\text{rec}}+nt_{\neq}+nt_{=} \\
&\qquad +(n+1)t_{\text{true}}+(3n+1)t_{\text{fst}}+nt_{\text{head}}+nt_{\text{tail}}+(n+1)t_{\text{nil?}}+nt_{+} \\
&\qquad +(n+2)t_{\text{rcase}}+\textstyle\sum_{i=0}^{n}t_{k_i} \\
V_{\text{rep}}(n,b) &= b
\end{aligned}$$

There is a greater cost in calculating $T_{\text{rep}}(n, \mathsf{ff})$ than $T_{\text{rep}}(n, \mathsf{tt})$ because when the continuation returns true on an empty string, the last repetition case only takes one branch, whereas if the continuation returns false, the last repetition case takes both branches. Otherwise the cost comes from a relatively straightforward case of examining each character in the string (and the continuation at each point); therefore it is not surprising that the cost is proportional to the length of the string when the cost of the continuation is itself constant.

Example 5: $r = (a_1|\ldots|a_n)^*$, $s = \langle a_n^m, l\rangle$, $k = \mathsf{initk}$.

In this example n is the number of choices in the regular expression, m is the the number of copies of the character a_n, and l is the number of characters read so far. For simplicity, we are assuming that all of the choices a_j are different from a_n for $j < n$, and we fixed the continuation. Thus if we allow n and m to vary, given that initk terminates, we can assume that there exist functions T_{rep_2} and V_{rep_2} such that

$$\mathsf{acc}(r, k, s) = [\![T_{\text{rep}_2}(n, m)]\!] \circ \eta \circ V_{\text{rep}_2}(n, m)$$

We can determine the values of T_{rep_2} by building recurrence equations from the following cases: when $m = 0$, when $m > 0$ and $n = 1$, and when $m > 0$ and $n > 1$. We similarly determine V_{rep_2} by induction on m using the same cases.

- Suppose that $m = 0$, i.e., s is snil. Then the continuation is true on s, so

$$\mathsf{acc}(r, k, s) = [\![4t_{\text{app}} + t_{\text{fst}} + t_{\text{nil?}} + t_{\text{rcase}} + t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt}$$

Therefore we know that

$$\begin{aligned} T_{\text{rep}_2}(n, 0) &= 4t_{\text{app}} + t_{\text{fst}} + t_{\text{nil?}} + t_{\text{rcase}} + t_{\text{true}} \\ V_{\text{rep}_2}(n, 0) &= \mathsf{tt} \end{aligned}$$

- Suppose that $m > 0$ but $n = 1$. This is the case where $s = a_n^m$ and r is $a_n{}^*$. In this case we match the conditions for Example 4 (initk satisfies the requirement on continuations). As for each $0 \leq i \leq m$,

$$\mathbf{apply}(\eta \circ \mathsf{initk}, a^i) = [\![t_{\text{app}} + t_{\text{fst}} + t_{\text{nil?}}]\!]$$

we immediately know that

$$\begin{aligned} T_{\text{rep}_2}(1, m) &= T_{\text{rep}}(m, \mathsf{tt}) \\ &= 3mt_{\text{snd}} + (12m + 2)t_{\text{app}} + 3mt_{\text{false}} + 2mt_{\text{rec}} + mt_{\neq} + mt_{=} + (m + 1)t_{\text{true}} \\ &\qquad + 3mt_{\text{fst}} + mt_{\text{head}} + mt_{\text{tail}} + mt_{\text{nil?}} + mt_{+} + (m + 1)t_{\text{rcase}} \\ &\qquad + \textstyle\sum_{i=0}^{m}(t_{\text{app}} + t_{\text{fst}} + t_{\text{nil?}}) \\ &= 3mt_{\text{snd}} + (13m + 3)t_{\text{app}} + 3mt_{\text{false}} + 2mt_{\text{rec}} + mt_{\neq} + mt_{=} + (m + 1)t_{\text{true}} \\ &\qquad + (4m + 1)t_{\text{fst}} + mt_{\text{head}} + mt_{\text{tail}} + (2m + 1)t_{\text{nil?}} + mt_{+} + (m + 1)t_{\text{rcase}} \\ V_{\text{rep}_2}(1, m) &= V_{\text{rep}}(m, \mathsf{tt}) \;=\; \mathsf{tt} \end{aligned}$$

- Suppose that $m > 0$ and $n > 1$. Then s is not the empty string, so

$$\mathbf{apply}(\eta \circ \mathsf{initk}, s) = [\![t_{\text{app}} + t_{\text{fst}} + t_{\text{nil?}}]\!] \circ \eta \circ \mathsf{ff}$$

thus

$$\begin{aligned}
\mathsf{acc}(r, k, s) &= [\![t_{\text{rcase}} + 3t_{\text{app}}]\!] \\
&\qquad \circ\ \mathbf{cond}(\mathbf{apply}(\eta \circ \mathsf{initk}, s))\ \mathbf{of} \\
&\qquad\quad \mathbf{True:}\quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
&\qquad\quad \mathbf{False:}\quad [\![2t_{\text{app}} + t_{\text{rec}} + t_{\text{false}}]\!] \circ \mathsf{acc}(r_0, \mathsf{checkk}(\mathsf{initk}, s, r_0), s) \\
&= [\![t_{\text{rcase}} + 6t_{\text{app}} + t_{\text{fst}} + t_{\text{nil?}} + t_{\text{rec}} + t_{\text{false}}]\!] \circ \mathsf{acc}(r_0, \mathsf{checkk}(\mathsf{initk}, s, r_0), s)
\end{aligned}$$

where $r_0 = a_1 | \ldots | a_n$, i.e., $r = r_0{}^*$.
Because k terminates, so does $\mathsf{checkk}(k, s, r_0)$. Furthermore, when we let $s' = \langle a_n^{m-1}, l+1 \rangle$ (that is, when s' is the tail of s) we have

$$\begin{aligned}
&\mathbf{apply}(\eta \circ \mathsf{checkk}(k, s, r_0), s') \\
&\quad = [\![2t_{\text{snd}} + 3t_{\text{app}} + t_{\text{rec}} + t_{\text{false}} + t_{\neq}]\!] \circ \mathsf{acc}(r, k, s') \\
&\quad = [\![2t_{\text{snd}} + 3t_{\text{app}} + t_{\text{rec}} + t_{\text{false}} + t_{\neq}]\!] \circ [\![T_{\text{rep}_2}(n, m-1)]\!] \circ \eta \circ V_{\text{rep}_2}(n, m-1)
\end{aligned}$$

By the induction hypothesis we know that $V_{\text{rep}_2}(n, m-1)$ is tt. Therefore all the assumptions needed to use the results of Example 1 are satisfied, where there are n choices, the index of the matching choice is also n, and

$$t = 2t_{\text{snd}} + 3t_{\text{app}} + t_{\text{rec}} + t_{\text{false}} + t_{\neq} + T_{\text{rep}_2}(n, m-1)$$

Therefore

$$\mathsf{acc}(r_0, \mathsf{checkk}(k, s, r_0), s) = [\![T_{\text{or}}(n, n)]\!] \circ \eta \circ \mathsf{tt}$$

where

$$\begin{aligned}
T_{\text{or}}(n, n) &= (11n - 6)t_{\text{app}} + (3n-2)t_{\text{false}} + (2n+1)t_{\text{fst}} + n(t_{\text{head}} + t_{\text{nil?}}) + (n-1)t_{\neq} \\
&\qquad + (2n-1)t_{\text{rcase}} + (2n-2)t_{\text{rec}} + t_{=} + t_{+} + t_{\text{snd}} + t_{\text{tail}} + t_{\text{true}} + t \\
&= (11n - 3)t_{\text{app}} + (3n-1)t_{\text{false}} + (2n+1)t_{\text{fst}} + n(t_{\text{head}} + t_{\text{nil?}} + t_{\neq}) \\
&\qquad + (2n-1)t_{\text{rcase}} + (2n-1)t_{\text{rec}} + t_{=} + t_{+} + 3t_{\text{snd}} + t_{\text{tail}} + t_{\text{true}} \\
&\qquad + T_{\text{rep}_2}(n, m-1)
\end{aligned}$$

From this we can derive the following equations:

$$\begin{aligned}
T_{\text{rep}_2}(n, m) &= (11n + 3)t_{\text{app}} + 3nt_{\text{false}} + (2n+2)t_{\text{fst}} + n(t_{\text{head}} + t_{\neq}) + (n+1)t_{\text{nil?}} \\
&\qquad + 2nt_{\text{rcase}} + 2nt_{\text{rec}} + t_{=} + t_{+} + 3t_{\text{snd}} + t_{\text{tail}} + t_{\text{true}} \\
&\qquad + T_{\text{rep}_2}(n, m-1) \\
V_{\text{rep}_2}(n, m) &= \mathsf{tt}
\end{aligned}$$

Thus we have shown that $V_{\text{rep}_2}(n, m) = \mathsf{tt}$ for all n and m, and have the following equations for $T_{\text{rep}_2}(n, m)$:

$$\begin{aligned}
T_{\text{rep}_2}(n, 0) &= 4t_{\text{app}} + t_{\text{fst}} + t_{\text{nil?}} + t_{\text{rcase}} + t_{\text{true}} \\
T_{\text{rep}_2}(1, m) &= 3mt_{\text{snd}} + (13m + 3)t_{\text{app}} + 3mt_{\text{false}} + 2mt_{\text{rec}} + mt_{\neq} + mt_{=} + (m+1)t_{\text{true}} \\
&\qquad + (4m+1)t_{\text{fst}} + mt_{\text{head}} + mt_{\text{tail}} + (2m+1)t_{\text{nil?}} + mt_{+} + (m+1)t_{\text{rcase}} \\
T_{\text{rep}_2}(n, m) &= (11n + 3)t_{\text{app}} + 3nt_{\text{false}} + (2n+2)t_{\text{fst}} + n(t_{\text{head}} + t_{\neq}) + (n+1)t_{\text{nil?}} \\
&\qquad + 2nt_{\text{rcase}} + 2nt_{\text{rec}} + t_{=} + t_{+} + 3t_{\text{snd}} + t_{\text{tail}} + t_{\text{true}} \\
&\qquad + T_{\text{rep}_2}(n, m-1)
\end{aligned}$$

To make the final calculation simpler, let $T'(n)$ be the following function:

$$\begin{aligned} T'(n) &= (11n+3)t_{\text{app}} + 3nt_{\text{false}} + (2n+2)t_{\text{fst}} + n(t_{\text{head}} + t_{\neq}) + (n+1)t_{\text{nil?}} \\ &\qquad + 2nt_{\text{rcase}} + 2nt_{\text{rec}} + t_{=} + t_{+} + 3t_{\text{snd}} + t_{\text{tail}} + t_{\text{true}} \end{aligned}$$

Thus $T_{\text{rep}_2}(n,m) = T'(n) + T_{\text{rep}_2}(n, m-1)$. From this we can easily see that

$$\begin{aligned} T_{\text{rep}_2}(1,m) &= 3mt_{\text{snd}} + (13m+3)t_{\text{app}} + 3mt_{\text{false}} + 2mt_{\text{rec}} + mt_{\neq} + mt_{=} + (m+1)t_{\text{true}} \\ &\qquad + (4m+1)t_{\text{fst}} + mt_{\text{head}} + mt_{\text{tail}} + (2m+1)t_{\text{nil?}} + mt_{+} + (m+1)t_{\text{rcase}} \\ T_{\text{rep}_2}(n,m) &= mT'(n) + 4t_{\text{app}} + t_{\text{fst}} + t_{\text{nil?}} + t_{\text{rcase}} + t_{\text{true}} \quad (n > 1) \end{aligned}$$

As the value of $T'(n)$ is linear in n, $T_{\text{rep}_2}(n,m)$ is proportional to nm, which means the cost of the pattern matching function in this case is $\mathcal{O}(nm)$. Intuitively, the cost $T'(n)$ corresponds to one round of checking all of the choices (plus some extra administrative cost) which must be done m times. The additional cost is the cost of successfully testing the empty string. The cost is different when $n = 1$ because then the regular expression is not a choice, so there is less administrative cost.

5.1.6 Complexity analysis

Determining the complexity for `accept` is significantly more difficult than it was for the examples in Chapter 3. Unlike those examples, we do not have a straightforward case for inductive formulas or proofs because `acc` is not always called recursively with a smaller regular expression or a smaller string. We can, however, discuss the program inductively; in all cases either the string becomes smaller or the string stays the same while the regular expression becomes smaller.

The use of continuations also complicates the problem of determining complexity. Because a continuation may differ throughout a computation, we cannot simply give a functional meaning for it as we did with the examples in Chapter 3. Furthermore, the number of times a continuation is called depends critically on the input.

We will not completely solve the first problem, but we can simplify the continuation problem. During the evaluation of `acc` on a regular expression r and string s, the program will call the continuation some number of times on various suffixes of s. If the result of the call is `true`, then program as a whole returns `true` with only an additional constant amount of work. If the result of the call is `false`, then the program may return `false`, or it may eventually call the continuation again. The program never returns `true` except by calling a continuation that returns `true`. If the continuation always returns `false`, then, because the program is guaranteed to terminate, the program eventually returns `false`. Therefore, for each r and s, there exists a list $[s_1, \ldots, s_n]$ of strings which represent, in order, the suffixes of s that a continuation will encounter, as long as it continues to return `false`. If a continuation returns `true` at some point, then it was called on s_1, $\ldots$, s_i for some $1 \leq i \leq n$, and it returned `false` for all but s_i. If two continuations give the same response on all the s_i, then the program cannot distinguish between them.

We can thus evaluate r and s with a continuation that mimics the original continuation but generates no cost and separately evaluate the original continuation on s_1 through s_i. We can thus separate the cost of the calls to the continuation from the cost of the rest of the evaluation.

We do not have any elements in our language to develop the mimicking continuations (although we could achieve a similar effect by adding special purpose constants). We can, however, define these continuations as morphisms in $\mathbf{PDom}^I$. When X is a discretely ordered set in $\mathbf{PDom}$, any function from X to a predomain Y is continuous and is thus a morphism in $\mathbf{PDom}$. Because each morphism from $\mathbf{1}$ to X is uniquely identified with an element of X, any function f from $\mathrm{Hom}(\mathbf{1}, X)$ to Y also uniquely defines a morphism f' from X to Y. Lastly this morphism in $\mathbf{PDom}$ defines a morphism (f', f') from id_X to id_Y. Because $\mathsf{List}\,(\mathrm{N})$ is discretely ordered and $\mathsf{List}^{\,I}(\mathrm{N}) = \mathsf{id}_{\mathsf{List}\,(\mathrm{N})}$,

$$\mathsf{List}^{\,I}(\mathrm{N})^I \times \mathbf{N}^I = \mathsf{id}_{\mathsf{List}\,(\mathrm{N})} \times \mathsf{id}_{\mathrm{N}} = \mathsf{id}_{\mathsf{List}\,(\mathrm{N})\times\mathrm{N}}$$

that is, the object $\mathsf{List}^{\,I}(\mathrm{N})^I \times \mathbf{N}^I$ in the arrow category is the identity morphism on $\mathsf{List}\,(\mathrm{N}) \times \mathbf{N}$. As this is the meaning of **string**, $\mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!] = \mathsf{id}_{\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{string}]\!]}$ and $\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{string}]\!]$ is discretely ordered. Similarly, we can show that $\mathcal{T}^{\mathrm{V}}[\![\mathbf{bool}]\!] = \mathsf{id}_{\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{bool}]\!]}$. Thus any function from extensional strings (i.e., morphisms from $\mathbf{1}$ to $\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{string}]\!]$) to $\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{bool}]\!]$ (which is equivalent to the set $\{\mathsf{tt}, \mathsf{ff}\}$) can be turned into a morphism from $\mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!]$ to $\mathcal{T}^{\mathrm{V}}[\![\mathbf{bool}]\!]$.

We can now define the mimicking continuation in terms of a function from extensional strings to the set $\{\mathsf{tt}, \mathsf{ff}\}$.

Definition 5.1.2 For any extensional string $s_E : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{string}]\!]$ and any morphism $z_E : \mathbf{1} \to L\mathbf{B}$ in $\mathbf{PDom}$, let $\mathsf{ekstr}(z_E, s_E) : \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{string}]\!] \to \mathbf{B}$ be the morphism equivalent to the function f_0 where $f_0(s) = \mathsf{ff}$ if either $s \neq s_E$ or $z_E = \mathsf{up} \circ \mathsf{ff}$, and $f_0(s) = \mathsf{tt}$ otherwise. For an intensional string $\mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!]$ and a morphism $z : \mathbf{1} \to C\mathcal{T}^{\mathrm{V}}[\![\mathbf{bool}]\!]$ in $\mathbf{PDom}^{\to}$, let

$$\mathsf{ikstr}(z, s) : \mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!] \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{bool}]\!] = (\mathsf{ekstr}(E(z), E(s)), \mathsf{ekstr}(E(z), E(s)))$$

be the lifted morphism in the intensional (arrow) category. To convert this to a continuation, let

$$\mathsf{kstr}(z, s) : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!] = \mathsf{curry}(\eta \circ \mathsf{ikstr}(z, s) \circ \pi_2)$$

Lastly, given a continuation $k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$, let

$$\mathsf{kkstr}(k, s) : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!] = \mathsf{kstr}(\mathbf{apply}(\eta \circ k, s), s)$$

Intuitively, $\mathsf{ekstr}(z_E, s_E)$ is equivalent to a simple function from extensional strings to boolean values that indicates if an input string s'_E is equal to s_E and, if so, that z_E is not known to be false. Thus $\mathsf{ekstr}(z_E, s_E)$ is a function that is false on all s'_E except s_E where it "mimics" z_E (except that $\perp$ values are converted to tt). The morphism $\mathsf{ikstr}(z, s)$ is simply the intensional version of $\mathsf{ekstr}(z_E, s_E)$, and the morphism $\mathsf{kstr}(z, s)$ is the curried version so that it is a continuation itself. The morphism $\mathsf{kkstr}(k, s)$ uses $\mathsf{kstr}(z, s)$ to define a continuation that is false everywhere except on s, where it "mimics" the behavior of k. For terminating k, the behavior is matched exactly (although without cost).

Definition 5.1.3 Given a continuation k from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$ and a string s from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!]$, let $\mathsf{kkstr}(k, s)$ as defined above be called the *mimicking continuation of k on s.*

The steps of the definition show that $\mathsf{kkstr}(k, s)$ always exists, and $\mathbf{apply}(\eta \circ k, s) = \mathbf{apply}(\eta \circ k', s)$ implies that $\mathsf{kkstr}(k, s) = \mathsf{kkstr}(k', s)$. Furthermore, for any strings s, s' and morphism z from $\mathbf{1}$ to $C\mathcal{T}^{\mathrm{V}}[\![\mathbf{bool}]\!]$,

$$\mathbf{apply}(\eta \circ \mathsf{kstr}(z, s), s') = \eta \circ \mathsf{ff}$$

when $s' \neq s$, and, for any cost t and any $b : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{bool}]\!]$,

$$\mathbf{apply}(\eta \circ \mathsf{kstr}([\![t]\!] \circ \eta \circ b, s), s) = \eta \circ b$$

Lastly, $\mathbf{apply}(\eta \circ \mathsf{kstr}(\perp, s), s) = \eta \circ \mathsf{tt}$.

Thus we know that $\mathsf{kstr}(z, s)$ always terminates, so let $\mathsf{kstrof}(z, s)$ be the morphism such that $\eta \circ \mathsf{kstrof}(z, s) = \mathbf{apply}(\eta \circ \mathsf{kstr}(z, s), s)$. We can do the same with $\mathsf{kkstr}(k, s)$, so let $\mathsf{kkstrof}(k, s)$ be $\mathsf{kstrof}(\mathbf{apply}(\eta \circ k, s), s)$, i.e.,

$$\eta \circ \mathsf{kkstrof}(k, s) = \mathbf{apply}(\eta \circ \mathsf{kkstr}(k, s))$$

Thus $\mathsf{kkstrof}(k, s)$ is the value portion of k applied to s.

The mimicking continuation becomes particularly useful when we note that the maximal list of suffixes $[s_1, \ldots, s_n]$ called by `acc` for a given regular expression r and string s, is not only the list of suffixes given to a continuation that always returns false, but also can be defined inductively from r and s. For example, if $r = r_1|r_2$, then `acc` first checks r_1 and then, if the result is false, checks r_2. Therefore the list of suffixes called by `acc` on $r_1|r_2$ and s should be the list of suffixes of r_1 and s appended by the list of suffixes of r_2 and s.

Definition 5.1.4 For any regular expression r and any string element $s : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{string}]\!]$, and any suffixes $\langle l, i\rangle$ and $\langle l', i'\rangle$ of s, let $S_{\mathrm{cont}}(r, s)$ and $S'_{\mathrm{cont}}(r, \langle l, i\rangle, \langle l', i\rangle)$ be defined by mutual induction as follows:

$$\begin{array}{rcll}
S_{\mathrm{cont}}(a, \mathsf{snil}(i)) & = & [\,] & \\
S_{\mathrm{cont}}(a, \mathsf{scons}(a, s)) & = & [s] & \\
S_{\mathrm{cont}}(a, \mathsf{scons}(a', s)) & = & [\,] & a' \neq a \\
S_{\mathrm{cont}}(r|q, s) & = & S_{\mathrm{cont}}(r, s)@S_{\mathrm{cont}}(q, s) & \\
S_{\mathrm{cont}}(rq, s) & = & S_{\mathrm{cont}}(q, s_1)@\ldots@S_{\mathrm{cont}}(q, s_n) & [s_1, \ldots, s_n] = S_{\mathrm{cont}}(r, s) \\
S_{\mathrm{cont}}(r^*, s) & = & [s]@S'_{\mathrm{cont}}(r, s, s_1)@\ldots@S'_{\mathrm{cont}}(r, s, s_n) & [s_1, \ldots, s_n] = S_{\mathrm{cont}}(r, s) \\
S'_{\mathrm{cont}}(r, \langle l, i\rangle, \langle l', i\rangle) & = & [\,] & \\
S'_{\mathrm{cont}}(r, \langle l, i\rangle, \langle l', i'\rangle) & = & S_{\mathrm{cont}}(r^*, \langle l', i\rangle) & i' \neq i
\end{array}$$

Definition 5.1.4 defines a function S_{cont} that takes a regular expression and a string and returns a list of suffixes; defined inductively on the regular expression and the string s. We will show this definition matches the intuitive concept of the list of maximal suffixes later (as a corollary to Theorem 5.1.5), in that the cost of $\mathsf{acc}(r, k, s)$ will be directly related to the cost of applying k to the elements of $S_{\mathrm{cont}}(r, s)$. We can, however, immediately check to see if the definition meets the intuitive concept. For example, as noted above $S_{\mathrm{cont}}(r|q, s) = S_{\mathrm{cont}}(r, s)@S_{\mathrm{cont}}(q, s)$. Similarly, for sequencing, given a regular expression rq, `acc` checks r with a continuation that checks q. Therefore if $[s_1, \ldots, s_n]$ is the list of suffixes from $S_{\mathrm{cont}}(r, s)$, then the list of suffixes for $S_{\mathrm{cont}}(rq, s)$ can be defined by replacing each s_i with the list of suffixes of s_i derived by q, i.e., $S_{\mathrm{cont}}(rq, s) = S_{\mathrm{cont}}(q, s_1)@\ldots@S_{\mathrm{cont}}(q, s_n)$. A similar result holds for r^*, except that we know that s is the first string in the list, as `acc`'s first action on r^* is to immediately call the continuation, and because of `strcheck`, we replace each s_i with $S'_{\mathrm{cont}}(r^*, s, s_i)$, where $S'_{\mathrm{cont}}(r, s, s')$ is the same as $S_{\mathrm{cont}}(r, s')$ except when $s = s'$, where it is the empty set. This corresponds to the action of `strcheck`, which returns false immediately if the strings have the same length. For $S_{\mathrm{cont}}(a, s)$, if the first character of s (assuming it has one) matches a, then the continuation is called on the tail of s' once; otherwise the program returns false without calling the continuation. Thus $S_{\mathrm{cont}}(a, s)$ is $[\,]$ except when $s = \mathsf{scons}(a, s')$, where it is $[s']$.

For a continuation k, we say that k is false on s if, for some cost t, $\mathbf{apply}(\eta \circ k, s) = [\![t]\!] \circ \eta \circ \mathsf{ff}$. Similarly, we say that k is true on s if, for some cost t, $\mathbf{apply}(\eta \circ k, s) = [\![t]\!] \circ \eta \circ \mathsf{tt}$. By adequacy (and because **PDom** is the underlying category), for all continuations k and strings s, either k is false on s, k is true on s, or $\mathbf{apply}(\eta \circ k, s) = \perp$.

Theorem 5.1.5 *For any continuation $k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$, any regular expression r, and any string s, let $[s_1, \ldots, s_n] = S_{\text{cont}}(r, s)$, and let i be the smallest integer where $1 \leq i \leq n$ and k is not false on s_i, or n if no such integer exists. Then one of the following two possibilities hold:*

- *$n = 0$ and there exists a cost t such that for all continuations k',*

$$\mathsf{acc}(r, k', s) = [\![t]\!] \circ \eta \circ \mathsf{ff}$$

- *$n > 0$ and*

$$\mathsf{acc}(r, k, s) = [\![t + \sum_{j=1}^{i-1} t_j]\!] \circ \mathbf{apply}(\eta \circ k, s_i)$$

  where $\mathsf{acc}(r, \mathsf{kkstr}(k, s_i), s) = [\![t]\!] \circ \eta \circ \mathsf{kkstrof}(k, s_i)$, and, for $1 \leq j < i$,

$$\mathbf{apply}(\eta \circ k, s_j) = [\![t_j]\!] \circ \eta \circ \mathsf{ff}$$

Proof. We give here a sketch of the proof; for a more formal proof see Appendix C.

From inspection of the program (and with induction) we can see that the following properties hold:

- `accept` can only return true if it calls a continuation that returns true
- If a continuation returns true (or fails to terminate), no more continuations are called.
- The only part of a continuation that is used is its result on the original string or its result on one of its suffixes.

With these facts it is clear that, assuming $S_{\text{cont}}(r, s)$ accurately describes the list of suffixes sent to the continuation, if $S_{\text{cont}}(r, s)$ is empty (i.e, the continuation is never called) then $\mathsf{acc}(r, k, s)$ must be false with a cost independent of k. Furthermore, the only difference between $\mathsf{acc}(r, k, s)$ and $\mathsf{acc}(r, \mathsf{kkstr}(k, s_i), s)$ is the cost of applying the continuations. Therefore the theorem holds.□

The formal proof shows that $S_{\text{cont}}(r, s)$ does match our intuitive definition, which can be seen by the result of the theorem when $n > 0$. Then the cost of the entire operation contains precisely one call to the continuation for each suffix s_i in the specified order until the continuation returns true.

5.1.7 More examples

With Theorem 5.1.5 we can better compare two regular expressions or calculate examples. Because $S_{\text{cont}}(r, s)$ is frequently easier to calculate than $\mathsf{acc}(r, k, s)$, we can sometimes use it to obtain a lower bound on the cost of a particular example. When k is false on all the elements of $S_{\text{cont}}(r, s)$, and produces some cost each time, then $\mathsf{acc}(r, k, s) = \Omega(l)$, where l is the length of $S_{\text{cont}}(r, s)$.

The theorem, however, is primarily used to compare similar regular expressions, as we can better separate the action of the regular expression from the action of the continuation. We can compare regular expressions r and r' by comparing $\mathbf{apply}(\mathcal{V}[\![\mathsf{acc}]\!], r)$ and $\mathbf{apply}(\mathcal{V}[\![\mathsf{acc}]\!], r')$ via $\preceq_{\mathbf{cont}\to\mathbf{string}\to\mathbf{bool}}$; however, because all continuations and strings are monotone, the relation $\mathbf{apply}(\mathcal{V}[\![\mathsf{acc}]\!], r) \preceq_{\mathbf{cont}\to\mathbf{string}\to\mathbf{bool}} \mathbf{apply}(\mathcal{V}[\![\mathsf{acc}]\!], r')$ holds if and only if for all continuations k and strings s, $\mathsf{acc}(r, k, s) \preceq_{\mathbf{bool}} \mathsf{acc}(r, k, s)$. Therefore

Definition 5.1.5 Let $r \preceq r'$ if for all terminating continuations k and strings s,

$$\mathsf{acc}(r, k, s) \preceq_{\mathbf{bool}} \mathsf{acc}(r', k, s)$$

Similarly, let $(r, s) \preceq (r', s')$ if for all terminating continuations k,

$$\mathsf{acc}(r, k, s) \preceq_{\mathbf{bool}} \mathsf{acc}(r', k, s')$$

The latter definition is useful for the cases where r may be faster than r' for some strings, but slower on others.

From the previous section we know that if k is a terminating continuation, then $\mathsf{acc}(r, k, s)$ must have the form $[\![t]\!] \circ \eta \circ b$, for some $t \in T$ and some $b : \mathbf{1} \to \mathbf{B}$. Alternatively, if k returns $\perp$ when called with some substring s_i, we know that the entire result becomes $\perp$. Therefore the behavior of $\mathsf{acc}(r, k, s)$ for all continuations is determined by its behavior on the terminating continuations.

The values of $S_{\text{cont}}(r, s)$ and $S_{\text{cont}}(r', s')$ can also tell us whether or not we can compare the two regular expressions with $\preceq$ at all, as seen in the following theorem:

Theorem 5.1.6 *Suppose that $(r, s) \preceq (r', s')$. Then a string s'' is in the sequence $S_{\text{cont}}(r, s)$ if and only if s'' is in the sequence $S_{\text{cont}}(r', s')$, i.e., $S_{\text{cont}}(r, s)$ and $S_{\text{cont}}(r', s')$ refer to the same* set *of strings (as opposed to the same sequence).*

Proof. Suppose that there exists a string s'' in the sequence of $S_{\text{cont}}(r, s)$ that is not in the sequence of $S_{\text{cont}}(r', s')$. Let $k = \mathsf{kstr}(\eta \circ \mathsf{tt}, s'')$, i.e., k is false on all strings except s'', where it is true (and the cost is always 0). Then by Theorem 5.1.5 there must exist costs t_1 and t_2 such that

$$\mathsf{acc}(r, k, s) = [\![t_1]\!] \circ \eta \circ \mathsf{tt} \quad \text{and} \quad \mathsf{acc}(r', k, s') = [\![t_2]\!] \circ \eta \circ \mathsf{ff}$$

Thus it cannot be true that $(r, s) \preceq (r', s')$. By a similar proof we can show that if there is a string s'' in the sequence $S_{\text{cont}}(r', s')$ that is not in $S_{\text{cont}}(r, s)$ then again it cannot hold that $(r, s) \preceq (r', s')$. Thus we know that both sequences refer to the same set of strings. □

To show a stronger result requires that we make some assumptions about the order $\preceq$. For example, it is reasonable to expect that for any costs t_1 and t_2, $t_1 \preceq t_1 + t_2$. This is true for just about any reasonable ordering we want on costs. We also require a reverse: if $t_2 \neq 0$, then $t_1 + t_2 \preceq t_1$ does *not* hold. This does *not* hold for the trivial ordering on costs, nor does it for any system where $t_1 + t_2 = t_1$ for some non-zero cost. However, for most cost systems, such a rule does hold. With that assumption, we have the following restriction on the relationship between $S_{\text{cont}}(r, s)$ and $S_{\text{cont}}(r', s')$ when $(r, s) \preceq (r', s')$:

Theorem 5.1.7 *Suppose that $(r, s) \preceq (r', s')$, that for all costs t_1 and t_2, if $t_2 \neq 0$, then $t_1 + t_2 \not\preceq t_1$, and that there exists a cost t_0 that is not 0. Then all of the following conditions hold:*

- *The set of strings in $S_{\text{cont}}(r, s)$ and $S_{\text{cont}}(r', s')$ are equal.*
- *The number of times a string s occurs in $S_{\text{cont}}(r, s)$ is less than or equal to the number of times it occurs in $S_{\text{cont}}(r', s')$*

Proof. The first condition is just a restatement of Theorem 5.1.6. For the second condition, assume it does not hold. Then there exists a string s_2 that occurs more times in $S_{\text{cont}}(r, s)$ than $S_{\text{cont}}(r', s')$, that is, for some $i > 0$ and $n > 0$, s_2 occurs n times in $S_{\text{cont}}(r', s')$ and $n + i$ times in $S_{\text{cont}}(r, s)$. For any cost t and string s_0, let $\mathsf{tcont}(t, s_0)$ be the continuation defined as follows:

$$\mathsf{tcont}(t, s_0) = \mathsf{cond} \circ \langle \mathsf{ikstr}(\eta \circ \mathsf{tt}, s_0) \circ \pi_2, [\![t]\!] \circ \eta \circ \mathsf{ff}, \eta \circ \mathsf{ff} \rangle$$

This means that for any string s_0',

$$\begin{aligned}\mathbf{apply}(\eta \circ \mathsf{tcont}(t, s_0), s_0') &= \mathbf{cond}(\mathsf{ikstr}(\eta \circ \mathsf{tt}, s_0) \circ s_0') \textbf{ of} \\ &\quad \textbf{True:} \quad [\![t]\!] \circ \eta \circ \mathsf{ff} \\ &\quad \textbf{False:} \quad \eta \circ \mathsf{ff} \\ &= \mathbf{cond}(\{s_0 = s_0'\}) \textbf{ of} \\ &\quad \textbf{True:} \quad [\![t]\!] \circ \eta \circ \mathsf{ff} \\ &\quad \textbf{False:} \quad \eta \circ \mathsf{ff}\end{aligned}$$

Thus $\mathsf{tcont}(t, s_0)$ is false on all strings and has 0 cost except on s_0, where the cost is t. Note that for all costs t and t' and all strings s_0 and s, $\mathsf{kkstr}(\mathsf{tcont}(t, s_0), s) = \mathsf{kkstr}(\mathsf{tcont}(t', s_0), s)$.

Because s_2 is in the sequence $S_{\text{cont}}(r, s)$ at least once, and because $\mathsf{tcont}(t, s_0)$ is always false, by Theorem 5.1.5 we know that there exists a cost t_1 such that for any cost t,

$$\mathsf{acc}(r, \mathsf{kkstr}(\mathsf{tcont}(t, s_2), s) = [\![t_1]\!] \circ \eta \circ \mathsf{ff}$$

and thus

$$\mathsf{acc}(r, \mathsf{tcont}(t, s_2), s) = [\![t_1 + (n+i)t]\!] \circ \eta \circ \mathsf{ff}$$

Similarly, there exists a cost t_2 such that for any cost t,

$$\mathsf{acc}(r', \mathsf{kkstr}(\mathsf{tcont}(t, s_2), s') = [\![t_2]\!] \circ \eta \circ \mathsf{ff}$$

and thus

$$\mathsf{acc}(r', \mathsf{tcont}(t, s_2), s') = [\![t_2 + nt]\!] \circ \eta \circ \mathsf{ff}$$

Let $t' = t_2 + t_0$. Then

$$\mathsf{acc}(r, \mathsf{tcont}(t', s_2), s) = [\![t_1 + (n+i)t_2 + (n+i)t_0]\!] \circ \eta \circ \mathsf{ff}$$

however,

$$\mathsf{acc}(r', \mathsf{tcont}(t', s_2), s') = [\![(n+1)t_2 + nt_0]\!] \circ \eta \circ \mathsf{ff}$$

As $i > 0$ and t_0 is not 0, we know that $t_1 + (n+i)t_2 + (n+i)t_0 \preceq (n+1)t_2 + nt_0$ does not hold. Thus $S_{\text{cont}}(r, s) \preceq S_{\text{cont}}(r', s')$ cannot hold either. As this is a contradiction, the third condition of the theorem must hold. □

For the rest of this section we will be assuming a particular (and common) ordering of the costs, i.e. $t_1 \preceq t_2$ precisely when there exists a t such that $t_1 + t = t_2$. For the integers this is the usual linear ordering. For the free commutative monoid over the set of cost constants (which is what we use in practice in this section), the ordering not only makes sense if one wants to interpret $\preceq$ as an indication of one program being faster than another, but also satisfies the assumptions made in Theorem 5.1.7.

Example 6: $r = (a^*)^*$, $s = \langle a^n, i \rangle$.

If $n = 0$, then $S_{\text{cont}}(a^*, s) = [s]$, and if $n > 0$, then $S_{\text{cont}}(a^*, s) = [s]@S_{\text{cont}}(a^*, \langle a^{n-1}, i+1 \rangle)$. Therefore $S_{\text{cont}}(a^*, s) = [\langle a^n, i \rangle, \ldots, \langle a^0, i+n \rangle]$. By the definition of S'_{cont}, we know that $S'_{\text{cont}}(a^*, s, \langle a^n, i \rangle) = []$, and for $k > 0$, $S'_{\text{cont}}(a^*, s, \langle a^{n-k}, i+k \rangle) = S_{\text{cont}}(r, \langle a^{n-k}, i+k \rangle)$.

Thus by the definition of S_{cont},

$$\begin{aligned}S_{\text{cont}}(r, s) &= [s]@S'_{\text{cont}}(a^*, \langle a^n, i \rangle)@\ldots@S'_{\text{cont}}(a^*, \langle a^0, i+n \rangle) \\ &= [s]@S_{\text{cont}}(r, \langle a^{n-1}, i+1 \rangle)@\ldots@S_{\text{cont}}(r, \langle a^0, i+n \rangle)\end{aligned}$$

Thus, as a function of n, the length of $S_{\text{cont}}(r,s)$ can be described by the following function:

$$\begin{aligned} L(0) &= 1 \\ L(n) &= 1 + \sum_{k=0}^{n-1} L(k) \end{aligned}$$

This means that the length of $S_{\text{cont}}(r,s)$ is 2^n, so for any continuation k that is false for each $\langle a^n, i\rangle, \ldots, \langle a^0, i+n\rangle$, $\mathsf{acc}(r,k,s)$ will have cost $\Omega(2^n)$. With a similar calculation we can show that $\mathsf{acc}(((a^*)^*)^*, k, s)$ has cost $\Omega(3^n)$.

Interestingly, the first $n+1$ values of $S_{\text{cont}}(r,s)$ are $\langle a^n, i\rangle, \ldots, \langle a^0, i+n\rangle$, that is, all the suffixes of s. Therefore if a continuation k is true on some element of $S_{\text{cont}}(r,s)$ is it true on one of the first $n+1$ elements. By examination of the code and the semantics it is clear that between calls to a continuation, the `accept` program performs a constant amount of work (i.e., a character comparison, a length comparison, and a single unrolling of the recursive call). Therefore $\mathsf{acc}(r,k,s)$ is $\mathcal{O}(n)$ in cost when k is constant in cost.

Example 7: Comparing $r_1|(r_2|r_3)$ and $(r_1|r_2)|r_3$:

Let r_1, r_2, r_3 be regular expressions, let k be a terminating continuation, and let s be a string.

Because k terminates, there exist costs t_1, t_2, and t_3, and morphisms b_1, b_2, b_3 from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}[\![\mathbf{bool}]\!]$ such that

$$\begin{aligned} \mathsf{acc}(r_1,k,s) &= [\![t_1]\!] \circ \eta \circ b_1 \\ \mathsf{acc}(r_2,k,s) &= [\![t_2]\!] \circ \eta \circ b_2 \\ \mathsf{acc}(r_3,k,s) &= [\![t_3]\!] \circ \eta \circ b_3 \end{aligned}$$

Then

$$\begin{aligned} \mathsf{acc}(r_1|(r_2|r_3),k,s) &= [\![5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}}]\!] \\ &\qquad \circ\ \mathbf{cond}([\![t_1]\!] \circ \eta \circ b_1)\ \mathbf{of} \\ &\qquad\quad \mathbf{True:}\quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathbf{tt} \\ &\qquad\quad \mathbf{False:}\quad [\![7t_{\text{app}} + t_{\text{false}} + t_{\text{rcase}} + 2t_{\text{rec}}]\!] \\ &\qquad\qquad \circ\ \mathbf{cond}([\![t_2]\!] \circ \eta \circ b_2)\ \mathbf{of} \\ &\qquad\qquad\quad \mathbf{True:}\quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathbf{tt} \\ &\qquad\qquad\quad \mathbf{False:}\quad [\![t_3 + 2t_{\text{app}} + t_{\text{false}} + t_{\text{rec}}]\!] \circ b_3 \\ &= [\![t_1 + 5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}}]\!] \\ &\qquad \circ\ \mathbf{cond}(\{b_1\})\ \mathbf{of} \\ &\qquad\quad \mathbf{True:}\quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathbf{tt} \\ &\qquad\quad \mathbf{False:}\quad [\![t_2 + 7t_{\text{app}} + t_{\text{false}} + t_{\text{rcase}} + 2t_{\text{rec}}]\!] \\ &\qquad\qquad \circ\ \mathbf{cond}(\{b_2\})\ \mathbf{of} \\ &\qquad\qquad\quad \mathbf{True:}\quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathbf{tt} \\ &\qquad\qquad\quad \mathbf{False:}\quad [\![t_3 + 2t_{\text{app}} + t_{\text{false}} + t_{\text{rec}}]\!] \circ b_3 \end{aligned}$$

and

$$\begin{aligned} \mathsf{acc}((r_1|r_2)|r_3,k,s) &= [\![t_1 + 10t_{\text{app}} + 2t_{\text{rcase}} + 2t_{\text{rec}}]\!] \\ &\qquad \circ\ \mathbf{cond}(z)\ \mathbf{of} \\ &\qquad\quad \mathbf{True:}\quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathbf{tt} \\ &\qquad\quad \mathbf{False:}\quad [\![t_3 + 2t_{\text{app}} + t_{\text{false}} + t_{\text{rec}}]\!] \circ b_3 \end{aligned}$$

where

$$z = \begin{array}{l}\textbf{cond}(\{b_1\})\ \textbf{of} \\ \quad \begin{array}{ll}\textbf{True:} & [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\ \textbf{False:} & [\![t_2 + 2t_{\text{app}} + t_{\text{false}} + t_{\text{rec}}]\!] \circ \eta \circ b_2\end{array}\end{array}$$

Now if $b_1 = \mathsf{tt}$, then

$$\mathsf{acc}(r_1|(r_2|r_3), k, s) = [\![t_1 + 5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}} + t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt}$$

while

$$\mathsf{acc}((r_1|r_2)|r_3, k, s) = [\![t_1 + 10t_{\text{app}} + 2t_{\text{rcase}} + 2t_{\text{rec}} + t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt}$$

Therefore right association is faster in this case. On the other hand, if $b_1 = \mathsf{ff}$, then

$$\begin{array}{rcl}\mathsf{acc}(r_1|(r_2|r_3), k, s) & = & [\![t_1 + t_2 + 12t_{\text{app}} + t_{\text{false}} + 2t_{\text{rcase}} + 3t_{\text{rec}}]\!] \\ & & \quad \circ\ \textbf{cond}(\{b_2\})\ \textbf{of} \\ & & \qquad \begin{array}{ll}\textbf{True:} & [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\ \textbf{False:} & [\![t_3 + 2t_{\text{app}} + t_{\text{false}} + t_{\text{rec}}]\!] \circ b_3\end{array}\end{array}$$

while

$$\begin{array}{rcl}\mathsf{acc}((r_1|r_2)|r_3, k, s) & = & [\![t_1 + t_2 + 12t_{\text{app}} + t_{\text{false}} + 2t_{\text{rcase}} + 3t_{\text{rec}}]\!] \\ & & \quad \circ\ \textbf{cond}(\{b_2\})\ \textbf{of} \\ & & \qquad \begin{array}{ll}\textbf{True:} & [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\ \textbf{False:} & [\![t_3 + 2t_{\text{app}} + t_{\text{false}} + t_{\text{rec}}]\!] \circ b_3\end{array}\end{array}$$

Thus when b_1 is false, right and left associations are identical. The reason why right association is faster when b_1 is true is because in that case the function never has to traverse the right-hand choice, but with left association (or when b_1 is false) the entire choice structure is traversed. The regular expressions are checked in the same order; the only possible change is the overhead in reaching the regular expressions.

Example 8: Comparing $r_1(r_2r_3)$ and $(r_1r_2)r_3$.

Let r_1, r_2, and r_3 be regular expressions and let s be a string. Then, for any continuation k,

$$\mathsf{acc}(r_1(r_2r_3), k, s) = [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}}]\!] \circ \mathsf{acc}(r_1, \mathsf{acck}(r_2r_3, k), s)$$

and

$$\mathsf{acc}((r_1r_2)r_3, k, s) = [\![14t_{\text{app}} + 2t_{\text{rcase}} + 4t_{\text{rec}}]\!] \circ \mathsf{acc}(r_1, \mathsf{acck}(r_2, \mathsf{acck}(r_3, k)), s)$$

For any suffix string s' (or s),

$$\begin{array}{rcl}\textbf{apply}(\eta \circ \mathsf{acck}(r_2r_3, k), s') & = & [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}}]\!] \circ \mathsf{acc}(r_2, \mathsf{acck}(r_3, k), s') \\ & = & [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}}]\!] \circ \textbf{apply}(\eta \circ \mathsf{acck}(r_2, \mathsf{acck}(r_3, k)), s')\end{array}$$

Thus the difference in the meanings of the two expressions is not which continuation is eventually called, but whether the cost $7t_{\text{app}}+t_{\text{rcase}}+2t_{\text{rec}}$ of traversing a sequence is counted in the beginning or in the continuation. Therefore the difference between the regular expressions is dependent on the number of times a continuation is called when evaluating r_1 on s. There are three cases to examine:

- Suppose that $S_{\text{cont}}(r_1, s)$ is empty. Then there exists a cost t such that for all continuations k', $\mathsf{acc}(r_1, k', s) = [\![t]\!] \circ \eta \circ \mathsf{ff}$. Thus

$$\begin{array}{rcl} \mathsf{acc}(r_1(r_2r_3), k, s) & = & [\![t + 7t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}}]\!] \circ \eta \circ \mathsf{ff} \\ & \preceq_{\mathbf{bool}} & [\![t + 14t_{\text{app}} + 2t_{\text{rcase}} + 4t_{\text{rec}}]\!] \circ \eta \circ \mathsf{ff} \\ & = & \mathsf{acc}((r_1r_2)r_3, k, s) \end{array}$$

  Therefore $(r_1(r_2r_3), s) \preceq ((r_1r_2)r_3, s)$

- Suppose that $S_{\text{cont}}(r_1, s) = [s']$, where s' is s or one of its suffixes. Then

$$\begin{array}{rcl} \mathsf{acc}(r_1(r_2r_3), k, s) & = & [\![t + 7t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}}]\!] \circ \mathbf{apply}(\eta \circ \mathsf{acck}(r_2r_3, k), s') \\ & = & [\![t + 14t_{\text{app}} + 2t_{\text{rcase}} + 2t_{\text{rec}}]\!] \circ \mathbf{apply}(\eta \circ \mathsf{acck}(r_2, \mathsf{acck}(r_3, k)), s') \\ & = & \mathsf{acc}((r_1r_2)r_3, k, s) \end{array}$$

  where by Theorem 5.1.5 there is a cost t such that

$$\begin{array}{rcl} \mathsf{acc}(r_1, \mathsf{kkstr}(\mathsf{acck}(r_2r_3, k), s'), s) & = & \mathsf{acc}(r_1, \mathsf{kkstr}(\mathsf{acck}(r_2, \mathsf{acck}(r_3, k)), s'), s) \\ & = & [\![t]\!] \circ \eta \circ \mathsf{kkstrof}(\mathsf{acck}(r_2r_3, k), s') \end{array}$$

  Because $S_{\text{cont}}(r_1, s)$ is a singleton, the program will call the continuation evaluating r_2 precisely once; therefore it does not matter if the overhead in exploring the sequence is handled before the continuation or in the continuation.

- Finally, suppose that $S_{\text{cont}}(r_1, s) = [s_1, \ldots, s_n]$, where $n > 1$. Let s_i be the first element such that $\mathbf{apply}(\eta \circ \mathsf{acck}(r_2r_3, k), s_i)$ is not false, or s_n if no such element exists. Let $s_{i'}$ be the first element such that

$$\mathbf{apply}(\eta \circ \mathsf{acck}(r_2, \mathsf{acck}(r_3, k)), s_i)$$

  is not false or s_n if no such element exists. As the two continuations are the same except for some additional initial costs, $s_i = s_{i'}$. Therefore, for each $1 \leq j < i$, there exists a cost t_j such that

$$\mathbf{apply}(\eta \circ \mathsf{acck}(r_2, \mathsf{acck}(r_3, k)), s_j) = [\![t_j]\!] \circ \eta \circ \mathsf{ff}$$

  Then, for each $1 \leq j < i$, $\mathbf{apply}(\eta \circ \mathsf{acck}(r_2r_3, k), s_j) = [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + t_i]\!] \circ \eta \circ \mathsf{ff}$, so

$$\begin{array}{rcl} \mathsf{acc}((r_1r_2)r_3, k, s) & = & [\![14t_{\text{app}} + 2t_{\text{rcase}} + 4t_{\text{rec}}]\!] \circ \mathsf{acc}(r_1, \mathsf{acck}(r_2, \mathsf{acck}(r_3, k)), s) \\ & = & [\![t + 14t_{\text{app}} + 2t_{\text{rcase}} + 4t_{\text{rec}} + \sum_{j=1}^{i-1} t_i]\!] \\ & & \quad \circ\, \mathbf{apply}(\eta \circ \mathsf{acck}(r_2, \mathsf{acck}(r_3, k)), s_i) \\ & \preceq_{\mathbf{bool}} & [\![t + 7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + \sum_{j=1}^{i-1}(7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + t_i)]\!] \\ & & \quad \circ\, [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}}]\!] \circ \mathbf{apply}(\eta \circ \mathsf{acck}(r_2, \mathsf{acck}(r_3, k)), s_i) \\ & = & \mathsf{acc}(r_1(r_2r_3), k, s) \end{array}$$

  Therefore, when the length of $S_{\text{cont}}(r_1, s)$ is greater than 1, $((r_1r_2)r_3, s) \preceq (r_1(r_2r_3), s)$. This reflects the property that, when a continuation is called more than once, the extra overhead in $r_1(r_2r_3)$ is also added more than once.

Example 9: Comparing $r(r_1|r_2)$ and $(rr_1)|(rr_2)$.

Let r, r_1, and r_2 be regular expressions. One would expect that $r(r_1|r_2) \preceq (rr_1)|(rr_2)$; this is not always the case, however. Let s be any string, and let $[s_1, \ldots, s_n] = S_{\text{cont}}(r, s)$. Then

$$S_{\text{cont}}(r(r_1|r_2), s) = S_{\text{cont}}(r_1, s_1)@S_{\text{cont}}(r_2, s_1)@\ldots@S_{\text{cont}}(r_1, s_n)@S_{\text{cont}}(r_2, s_n)$$

but

$$S_{\text{cont}}(rr_1|rr_2, s) = S_{\text{cont}}(r_1, s_1)@\ldots@S_{\text{cont}}(r_1, s_n)@S_{\text{cont}}(r_2, s_1)@\ldots S_{\text{cont}}(r_2, s_n)$$

Therefore we immediately know that we cannot compare $r(r_1|r_2)$ and $rr_1|rr_2$ for all strings. For example, for any continuation k and string s,

$$\mathsf{acc}(a(b|c), k, \mathsf{scons}(d, s)) \preceq_{\mathbf{bool}} \mathsf{acc}(ab|ac, k, \mathsf{scons}(d, s))$$

but, for any integer i,

$$\mathsf{acc}(a^*bc|a^*ab, \mathsf{initk}, \langle [a, b, c], i\rangle) \preceq_{\mathbf{bool}} \mathsf{acc}(a^*(bc|ab), \mathsf{initk}, \langle [a, b, c], i\rangle)$$

because the former only needs to call initk once.

Even if $S_{\text{cont}}(r(r_1|r_2), s)$ and $S_{\text{cont}}(rr_1|rr_2, s)$ happen to be equal, we cannot guarantee that $(r(r_1|r_2), s) \preceq (rr_1|rr_2, s)$. For example,

$$S_{\text{cont}}(a^*bc|a^*abd, \langle [a, b, c], i\rangle) = S_{\text{cont}}(a^*(bc|abd), \langle [a, b, c], i\rangle) = [\mathsf{snil}(i + 3)]$$

because each version has a branch that requires all three characters (and the other branches fail), but

$$\mathsf{acc}(a^*bc|a^*abd, \mathsf{initk}, \langle [a, b, c], i\rangle) \preceq_{\mathbf{bool}} \mathsf{acc}(a^*(bc|abd), \mathsf{initk}, \langle [a, b, c], i\rangle)$$

The extra cost comes from having to test abd (in the initial case where a^* is being tested against the empty string) unnecessarily.

There are, however, cases where $S_{\text{cont}}(rr_1|rr_2, s) \preceq S_{\text{cont}}(r(r_1|r_2), s)$. For example, suppose that $S_{\text{cont}}(r, s) = [\,]$. This means that the program fails without ever reaching r_1 or r_2. Therefore, as, for all k,

$$\mathsf{acc}(r(r_1|r_2), k, s) = [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}}]\!] \circ \mathsf{acc}(r, \mathsf{acck}(r_1|r_2, k), s)$$

and

$$\begin{array}{rcl} \mathsf{acc}(rr_1|rr_2, k, s) & = & [\![12t_{\text{app}} + 2t_{\text{rcase}} + 3t_{\text{rec}}]\!] \\ & & \quad \circ\ \mathbf{cond}(\mathsf{acc}(r, \mathsf{acck}(r_1, k), s))\ \mathbf{of} \\ & & \qquad \mathbf{True:}\quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\ & & \qquad \mathbf{False:}\quad [\![9t_{\text{app}} + t_{\text{false}} + t_{\text{rcase}} + 3t_{\text{rec}}]\!] \circ \mathsf{acc}(r, \mathsf{acck}(r_2, k), s) \end{array}$$

we know immediately that $(r(r_1|r_2), s) \preceq (rr_1|rr_2, s)$ because $\mathsf{acc}(r, k, s)$ is always false with a cost independent of the continuation. In this case not only is there the additional overhead of exploring the choice operator, but r is checked twice.

Next suppose that $S_{\text{cont}}(r, s) = [s']$, i.e., r matches some prefix of s and has no other matching choices. Then there exist costs t_1 and t_2 such that

$$\mathsf{acc}(r, \mathsf{kstr}(\eta \circ \mathsf{tt}, s), s) = [\![t_1]\!] \circ \eta \circ \mathsf{tt} \quad \text{and} \quad \mathsf{acc}(r, \mathsf{kstr}(\eta \circ \mathsf{ff}, s), s) = [\![t_2]\!] \circ \eta \circ \mathsf{ff}$$

Therefore, if $\mathsf{acck}(r_1, k)$ is true on s' with cost t_1' (i.e., $\mathbf{apply}(\eta \circ \mathsf{acck}(r_1, k), s') = [\![t_1']\!] \circ \eta \circ \mathsf{tt}$), then, by Theorem 5.1.5,

$$\begin{aligned}
\mathsf{acc}(r, \mathsf{acck}(r_1, k), s) &= [\![t_1]\!] \circ \mathbf{apply}(\eta \circ \mathsf{acck}(r_1, k), s') \\
&= [\![t_1' + t_1]\!] \circ \eta \circ \mathsf{tt}
\end{aligned}$$

and

$$\begin{aligned}
\mathsf{acc}(r, \mathsf{acck}(r_1|r_2, k), s) &= [\![t_1]\!] \circ \mathbf{apply}(\eta \circ \mathsf{acck}(r_1|r_2, k), s') \\
&= [\![5t_{\mathrm{app}} + t_{\mathrm{rcase}} + t_{\mathrm{rec}} + t_1]\!] \\
&\qquad \circ\ \mathbf{cond}(\mathsf{acc}(r_1, k, s'))\ \mathbf{of} \\
&\qquad\qquad \mathbf{True:}\quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
&\qquad\qquad \mathbf{False:}\quad [\![2t_{\mathrm{app}} + t_{\mathrm{false}} + t_{\mathrm{rec}}]\!] \circ \mathsf{acc}(r_2, k, s') \\
&= [\![5t_{\mathrm{app}} + t_{\mathrm{rcase}} + t_{\mathrm{rec}} + t_{\mathrm{true}} + t_1' + t_1]\!] \circ \eta \circ \mathsf{tt}
\end{aligned}$$

Therefore

$$\begin{aligned}
\mathsf{acc}(r(r_1|r_2), k, s) &= [\![7t_{\mathrm{app}} + t_{\mathrm{rcase}} + 2t_{\mathrm{rec}}]\!] \circ \mathsf{acc}(r, \mathsf{acck}(r_1|r_2, k), s) \\
&= [\![12t_{\mathrm{app}} + 2t_{\mathrm{rcase}} + 3t_{\mathrm{rec}} + t_{\mathrm{true}} + t_1' + t_1]\!] \circ \eta \circ \mathsf{tt} \\
&= [\![12t_{\mathrm{app}} + 2t_{\mathrm{rcase}} + 3t_{\mathrm{rec}} + t_1]\!] \\
&\qquad \circ\ \mathbf{cond}(\mathsf{acc}(r_1, k, s'))\ \mathbf{of} \\
&\qquad\qquad \mathbf{True:}\quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
&\qquad\qquad \mathbf{False:}\quad [\![9t_{\mathrm{app}} + t_{\mathrm{false}} + t_{\mathrm{rcase}} + 3t_{\mathrm{rec}} + t_1]\!] \circ \mathsf{acc}(r_2, k, s') \\
&= \mathsf{acc}(rr_1|rr_2, k, s)
\end{aligned}$$

In this case they are equal, because the second choice operator is never needed (in either case) and the overhead in exploring the regular expression is also the same.

If, on the other hand, $\mathsf{acck}(r_1, k)$ is false on s' with a cost t_1', then

$$\begin{aligned}
&\mathbf{apply}(\eta \circ \mathsf{acck}(r_1|r_2, k), s') \\
&\quad = [\![5t_{\mathrm{app}} + t_{\mathrm{rcase}} + t_{\mathrm{rec}}]\!] \circ\ \mathbf{cond}(\mathsf{acc}(r_1, k, s'))\ \mathbf{of} \\
&\qquad\qquad \mathbf{True:}\quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
&\qquad\qquad \mathbf{False:}\quad [\![2t_{\mathrm{app}} + t_{\mathrm{false}} + t_{\mathrm{rec}}]\!] \circ \mathsf{acc}(r_2, k, s') \\
&\quad = [\![7t_{\mathrm{app}} + t_{\mathrm{rcase}} + 2t_{\mathrm{rec}} + t_{\mathrm{false}} + t_1']\!] \circ \mathsf{acc}(r_2, k, s')
\end{aligned}$$

so we know that

$$\begin{aligned}
&\mathsf{acc}(r(r_1|r_2), k, s) \\
&\quad = [\![7t_{\mathrm{app}} + t_{\mathrm{rcase}} + 2t_{\mathrm{rec}} + t]\!] \circ \mathbf{apply}(\eta \circ \mathsf{acck}(r_1|r_2, k), s') \\
&\quad = [\![14t_{\mathrm{app}} + 2t_{\mathrm{rcase}} + 4t_{\mathrm{rec}} + t_{\mathrm{false}} + t + t_1']\!] \circ \mathsf{acc}(r_2, k, s') \\
&\quad \preceq_{\mathbf{bool}} [\![21t_{\mathrm{app}} + 3t_{\mathrm{rcase}} + 6t_{\mathrm{rec}} + t_{\mathrm{false}} + 2t + t_1']\!] \circ \mathsf{acc}(r_2, k, s') \\
&\quad = [\![12t_{\mathrm{app}} + 2t_{\mathrm{rcase}} + 3t_{\mathrm{rec}} + t]\!] \\
&\qquad \circ\ \mathbf{cond}(\mathsf{acc}(r_1, k, s'))\ \mathbf{of} \\
&\qquad\qquad \mathbf{True:}\quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
&\qquad\qquad \mathbf{False:}\quad [\![9t_{\mathrm{app}} + t_{\mathrm{false}} + t_{\mathrm{rcase}} + 3t_{\mathrm{rec}} + t]\!] \circ \mathsf{acc}(r_2, k, s') \\
&\quad = \mathsf{acc}(rr_1|rr_2, k, s)
\end{aligned}$$

where $t = t_1$ if $\mathsf{acck}(r_2, k)$ is false on s', and $t = t_2$ otherwise. Therefore

$$(r(r_1|r_2), s) \preceq (rr_1|rr_2, s)$$

Again with $rr_1|rr_2$, the extra checks on r make it slower.

Even when we cannot compare $r(r_1|r_2)$ and $rr_1|rr_2$ for all continuations, we can compare them for select continuations. For example, if s_1 is the first string in $S_{\text{cont}}(r, s)$ and k is true on some suffix s' of $S_{\text{cont}}(r_1, s_1)$, then the values of $\mathsf{acc}(r(r_1|r_2), k, s)$ and $\mathsf{acc}(rr_1|rr_2, k, s)$ are similar to the values in the previous example, and $\mathsf{acc}(r(r_1|r_2), k, s) \preceq_{\mathbf{bool}} \mathsf{acc}(rr_1|rr_2, k, s)$.

We also can predict the values of $\mathsf{acc}(r(r_1|r_2), k, s)$ and $\mathsf{acc}(rr_1|rr_2, k, s)$ when k is false everywhere except possibly in $S_{\text{cont}}(r_2, s_n)$. In those cases, we know that, for $1 \leq i < n$, $\mathsf{acc}(r_1, k, s_i)$ and $\mathsf{acc}(r_2, k, s_i)$ are both false, which means that there exist costs t'_i and t''_i such that $\mathsf{acc}(r_1, k, s_i) = [\![t'_i]\!] \circ \eta \circ \mathsf{ff}$ and $\mathsf{acc}(r_2, k, s_i) = [\![t''_i]\!] \circ \eta \circ \mathsf{ff}$. Furthermore there exists a cost t'_n such that $\mathsf{acc}(r_1, k, s_n) = [\![t'_n]\!] \circ \eta \circ \mathsf{ff}$.

Let b be tt if k is true in some suffix s' in $S_{\text{cont}}(r_2, s_n)$ and ff otherwise. Then there exists a cost $[\![t''_n]\!]$ such that $\mathsf{acc}(r_2, k, s_n) = [\![t''_n]\!] \circ \eta \circ b$. Therefore, for $1 \leq i \leq n$,

$$\begin{aligned}
\mathsf{acc}(r_1|r_2, k, s_i) = \; & [\![5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}}]\!] \circ \; \mathbf{cond}(\mathsf{acc}(r_1, k, s_i)) \; \mathbf{of} \\
& \qquad\qquad \mathbf{True:} \quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
& \qquad\qquad \mathbf{False:} \quad [\![2t_{\text{app}} + t_{\text{false}} + t_{\text{rec}}]\!] \circ \mathsf{acc}(r_2, k, s_i) \\
= \; & [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}} + t_{\text{false}} + t'_i + t''_i]\!] \circ \eta \circ b
\end{aligned}$$

By Theorem 5.1.5 there exist costs t_1 and t_2 such that $\mathsf{acc}(r, \mathsf{kstr}(\eta \circ \mathsf{ff}, s_n), s) = [\![t_1]\!] \circ \eta \circ \mathsf{ff}$, and $\mathsf{acc}(r, \mathsf{kstr}(\eta \circ b, s_n), s) = [\![t_2]\!] \circ \eta \circ b$. Therefore

$$\mathsf{acc}(r, \mathsf{acck}(r_1, k), s) = [\![t_1 + \sum_{i=1}^{n} t'_i]\!] \circ \eta \circ \mathsf{ff},$$

$$\mathsf{acc}(r, \mathsf{acck}(r_2, k), s) = [\![t_2 + \sum_{i=1}^{n} t''_i]\!] \circ \eta \circ b,$$

and

$$\mathsf{acc}(r, \mathsf{acck}(r_1|r_2, k), s) = [\![t_2 + 7nt_{\text{app}} + nt_{\text{rcase}} + 2nt_{\text{rec}} + nt_{\text{false}} + \sum_{i=1}^{n}(t'_i + t''_i)]\!] \circ \eta \circ b$$

so

$$\begin{aligned}
& \mathsf{acc}(r(r_1|r_2), k, s) \\
& \quad = [\![7t_{\text{app}} + t_{\text{rcase}} + 2t_{\text{rec}}]\!] \circ \mathsf{acc}(r, \mathsf{acck}(r_1|r_2, k), s) \\
& \quad = [\![t_2 + 7(n+1)t_{\text{app}} + (n+1)t_{\text{rcase}} + 2(n+1)t_{\text{rec}} + nt_{\text{false}} + \textstyle\sum_{i=1}^{n}(t'_i + t''_i)]\!] \circ \eta \circ b
\end{aligned}$$

and

$$\begin{aligned}
& \mathsf{acc}(rr_1|rr_2, k, s) \\
& \quad = [\![12t_{\text{app}} + 2t_{\text{rcase}} + 3t_{\text{rec}}]\!] \\
& \qquad\quad \circ \; \mathbf{cond}(\mathsf{acc}(r, \mathsf{acck}(r_1, k), s)) \; \mathbf{of} \\
& \qquad\qquad \mathbf{True:} \quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
& \qquad\qquad \mathbf{False:} \quad [\![9t_{\text{app}} + t_{\text{false}} + t_{\text{rcase}} + 3t_{\text{rec}}]\!] \circ \mathsf{acc}(r, \mathsf{acck}(r_2, k), s) \\
& \quad = [\![21t_{\text{app}} + 3t_{\text{rcase}} + 6t_{\text{rec}} + t_{\text{false}} + t_1 + t_2 + \textstyle\sum_{i=1}^{n}(t'_i + t''_i)]\!] \circ \eta \circ b
\end{aligned}$$

To determine which regular expression is faster, we need to know if t_1 is greater or less than $(7n-14)t_{\text{app}} + (n-2)t_{\text{rcase}} + (n-1)t_{\text{false}} + (2n-4)t_{\text{rec}}$. The cost t_1 is approximately n times the average cost between calls to the continuations. This average cost, therefore, needs to be less than $7t_{\mathrm{app}} + t_{\mathrm{rcase}} + t_{\mathrm{false}} + 2t_{\mathrm{rec}}$. The only cases where $S_{\mathrm{cont}}(r, s)$ is greater than 1 are when r contains either choice or repetition. In either case t is at least $7t_{\mathrm{app}} + t_{\mathrm{rcase}} + t_{\mathrm{false}} + 2t_{\mathrm{rec}}$, so $r(r_1|r_2)$ will be faster. Note that n must be at least 2 for the calculated cost to be non-negative; this is another indication that for $n = 1$, $(r(r_1|r_2), s) \preceq (rr_1|rr_2, s)$.

5.2 Conclusion

The `accept` program is a relatively simple program that uses continuations. It has an advantage over the traditional regular expression matching program in that it requires little set up time. It has a disadvantage, however, that for some regular expressions the cost of evaluating the expression is slower; sometimes significantly slower. Fortunately for the only regular expressions found that have an exponential cost (such as $(a^*)^*$) there exist simpler expressions with non-exponential cost representing the same set of strings.

Some of the complexities found for particular regular expression came as no surprise. For example, the cost of matching $a_1|\ldots|a_n$ or $a_1a_2\ldots a_n$ are both linear in n, while matching $(a_1|\ldots|a_n)^*$ against a string of m a_n's has a complexity of $\mathcal{O}(nm)$. Similarly we found that the only difference between $r_1|(r_2|r_3)$ or $(r_1|r_2)|r_3$ is possibly the time it takes to reach r_1.

Several of the complexity properties, however, were non-intuitive and required careful analysis to discover them. For example until we compared $r(r_1|r_2)$ and $rr_1|rr_2$ with the help of Theorem 5.1.5 we assumed that the former expression was always faster. The analysis not only proved differently, but pinpointed the cases where the unfactored expression was faster. Similarly, the difference between $r_1(r_2r_3)$ and $(r_1r_2)r_3$ was greater than expected; with each version being potentially faster depending on the value of S_{cont}. Lastly, our analysis not only showed that there were expressions that might require exponential cost to match but also showed which strings were being rechecked. This analysis was made possible by both from the treatment of functional types and the ability to mathematically substitute values and thus enable us to separate the cost of the continuations from the cost of the regular expression itself.

Chapter 6

Conclusions and Future Work

In this dissertation we create a technique for building intensional semantics for functional programs using either call-by-value or call-by-name evaluation strategies. We start with standard call-by-value and call-by-name denotational and operational semantics. The depiction of the denotational semantics is slightly unusual in that we express the treatment of nontermination explicitly through the use of the lifting monad. From these standard extensional semantics we create intensional semantics. For the intensional operational semantics, we simply label the transitions with costs that relate to the steps being taken during execution. For the intensional denotational semantics we first create a monad-like structure called a cost structure that is capable of representing the cost of evaluation. We then replace the lifting monad with the cost structure and add constants costs as needed to correctly represent the cost of evaluation.

We show that the computation of cost is reasonable by proving that the intensional semantic functions are sound and adequate relative to an operational semantics in which the relationship between the costs and the steps of the evaluation is clear. We also show that the intensional semantic functions are consistent with the extensional semantics in that we can easily recover the extensional denotational semantics from the intensional. Additionally, we show that there are non-trivial cost structures and that it is possible to derive meaningful information about the behavior of functions using the intensional semantics. We show this for some simple examples for both the call-by-name and call-by-value semantics and show results for a more complicated example (using continuations) using the call-by-value semantics.

The technique used to derive the intensional semantics from the extensional is robust; with virtually the same technique we are able to derive intensional semantics for both the call-by-value and call-by-name semantics. We also can easily add several data types and constants to the language without losing soundness or adequacy. These additional data types include many of the basic structures used for many kinds of defined data types seen in real programs; we believe that most data types used in purely applicative programs (i.e., those that do not involve state changes such as references) should be analyzable by our semantic functions.

There are two factors aiding in this robustness. First, we based our cost structure on monads. Monads have been much studied in relationship to computer science (see [26], [48], [46]) and have been shown to be a useful general method for adding structure to semantics or to data types. In this case we did not need all the properties of a monad to prove the desired results needed from the cost structure; however, by defining proper and strong cost structures we showed that a cost structure can have all the properties of a monad or a strong monad. In particular, the non-trivial example we created for a cost structure did satisfy all the properties of a strong cost structure and thus was also a strong monad.

Second, because cost analysis and strictness analysis are closely connected (see [45]), we tightly couple our semantics with the lifting monad. As a result we also gain the ability to build our intensional semantics on top of a well-known extensional semantics with explicit strictness properties. Furthermore, it is often easy to devise ways to add many additional constants or languages because there is usually a straightforward technique to convert their extensional semantic properties to the intensional (such as converting the lifting monad to a cost structure and adding constant costs where needed). Thus we have a greater guarantee of both the appropriateness and the robustness of our technique.

What is new about our technique is this combination of internal costs through a monadic structure plus the tight coupling with the lifting monad. Other techniques that have used monads to add cost ([12], [41]) tend to ignore strictness properties, making the resulting semantics less obviously correct and more difficult to generalize. Some earlier approaches that do pay attention to strictness properties ([37], [6]) handle costs externally and thus do not appear to generalize naturally to higher order types.

The semantic functions defined in this dissertation give us the ability to analyze complexity for call-by-value and call-by-name programs. Because of the close relationship between the call-by-name and call-by-need evaluation strategies, we can also analyze complexity for some call-by-need programs and approximate costs for others. In some cases, where the complexity inherent in high order or lazy programs is sufficiently complicated so that normal complexity analysis becomes difficult, our semantics is particularly useful. This was particularly noticeable when analyzing the complexity of the continuation-style pattern matching program, whose analysis obviously required the formulation of intensional properties for continuations. We are able to show complexity results that are not necessarily intuitive. For example, it was sometimes difficult to tell in advance which of two extensionally equivalent regular expressions (such as $r_1(r_2|r_3)$ and $r_1r_2|r_1r_3$) are used more efficiently in the pattern matching program. Furthermore, it was not intuitively obvious that there would be regular expressions such that the program had exponential time relative to the length of the string to be matched.

6.1 Related Work

6.1.1 David Sands: Calculi for Time Analysis of Functional Programs

In Sands' thesis, he calculated cost (in his case, the number of function calls) by, for each function, creating a separate *cost function* in the language that calculated the cost instead of the final value. For example, a program such as

```
map f x = if (null x) then nil
          else (cons (f (hd x)) (map f (tl x)))
```

would have the following cost function:

```
cmap f x = 1 + if (null x) then 0
           else (f c@ (hd x)) + (cmap f (tl x))
```

where `c@` is an abbreviations for applying the cost function of the application function `apply f x = f(x)`. The intensional meaning of a function is then taken to be the standard extensional meaning of the function paired with the meaning of the cost function, plus some arity information needed to keep track of higher order types.

For first-order call-by-value functions, the difference between his system and the one in this dissertation are relatively minor. His system is flexible enough to count the types of costs we used (and vice versa), with the exception that, because his language structure does not distinguish between recursive and non-recursive functions, Sands' system cannot count the number of recursive calls as opposed to function calls in general. If, however, he had used explicit recursion in his language definition then such information would probably be available.

For higher-order call-by-value, the difference is greater, but still not substantial. Sands distinguished between the application of a curried function that still requires arguments from ones that do not (as we do for primitive constants). He was thus able to give different costs for applying the two types of functions. Our system does not currently make such a distinction, although it could be augmented to do so with a change to the operational semantics so that the arity of a λ-expression is used to generate different costs of application. Outside of those differences, again either method gives the same results for the same set of functions.

The real distinction between the two frameworks is when we look at lazy data structures (which our system has in the call-by-name system). Because Sands added costs by modifying functions, his system does not appear to generalize easily to the case where computation is delayed because part of a data structure is not evaluated. Instead his method involved the evaluation of a function relative to a context, based on Wadler's projections ([44]), that describes how much of the result is going to be needed. A *projection* is an idempotent function f such that $f(x) \sqsubseteq x$. For the purposes of Sands' evaluation, a projection removes from a value information that will not be needed. For example, if a projection returns $\perp$ we know that the value will not be used, so we do not need to include its cost.

The problem with this system occurs when converting the projection for a function call to the projections needed for each of the arguments. There is no computable way to do so, which can be a problem even with the primitive (constant) functions (although for most of them, useful solutions are known), but is a serious problem for higher-order types. Because of this, Sands could only approximate the cost calculated, and in some cases those approximate costs were arbitrarily poor (and in a few cases, the automatic cost calculation led to non-termination even though the actual function terminates).

6.1.2 Jon Shultis: On the Complexity of Higher-Order Program

Shultis ([41]) wrote a technical report discussing a higher-order call-by-value semantics that included cost. He wrote a denotational semantics for a simple call-by-value language which can be used to evaluate not only the value but the *toll* associated with an expression. For expressions of non-functional type, a toll is similar to a cost in our system (except that Shultis counted the use of values but not application or recursion). For higher order functions, tolls also resemble what we called internal cost, i.e., a toll included the cost of applying a function as well as the cost of evaluating the function. These tolls most closely resemble cost functions in the style of Sands except that they were presented by Shultis as a mathematical objects rather than as functions written in the programming language itself. Therefore if `Cost` is a set of costs (which in Shultis' case meant integers), and `Value` is the set of all possible values (integers, lists, functions, etc), then a function of type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$ therefore had n tolls associated with it: $\mathtt{t}^0 : \mathtt{Cost}$, $\mathtt{t}^1 : \mathtt{Value} \rightarrow \mathtt{Cost}$, the cost of applying the function to the first argument, and so forth up to $\mathtt{t}^n$. The use of tolls was the primary accomplishment of the paper; Shultis explicitly distinguished between the cost of evaluating an expression and the cost of using the expression in future evaluations, analogous to our distinction between external and internal cost.

In practice, Shultis' system gives similar costs as ours (or Sands) for most functions. For example, his system gives the cost of `twice a b` as (using his notation)

$$5 + \mathtt{t}_a^0 + \mathtt{t}_b^0 + \mathtt{t}_a^1(v_b) + \mathtt{t}_a^1(v_a(v_b))$$

where for $x = a$ or $x = b$, v_x is the value of x, $\mathtt{t}_x^0$ is the cost of evaluating x, and $\mathtt{t}_a^1(x)$ is the cost of applying v_a to a value. The constant 5 is derived primarily by evaluating λ-expressions. This is essentially the same as the cost we get for the function, except that instead of the constant 5 we have the cost of two applications. The examples in his paper all have the same asymptotic costs as in our system; however, the constants involved are quite different due to the different choice of base costs.

The work described in [41], however, was only preliminary. There was no attempt to compare his semantics with any operational model. Shultis did implement his functions in a program and used that to check some of the more complicated calculations, but this was a simple implementation of his denotational semantics, not an operational definition. In fact, his system is not sound in comparison to the operational models used by us or by Sands because his choice of costs included non-zero costs for values. Even against an operational semantics that distinguishes between values and expressions that immediately return values, his system adds cost for each re-evaluation of a value, so it would not be consistent with that operational model.

Also, as his focus was on higher-order types, Shultis did not have a method for handling costs related to call-by-name evaluation strategies or with lazy data structures (where there is another form of internal cost).

6.1.3 Douglas Gurr: Semantic Frameworks for Complexity

Douglas Gurr in ([12]) worked on a system of adding complexity to already existing semantic systems. He does this primarily using two techniques. The first technique uses the product monad to add complexity to a given language calculus. His use of the product monad closely resembles our use of cost structures; the differences are minor (such as differences between how costs are introduced). The primary difference is in how they were used. We were primarily concerned with internal costs and separability; Gurr was more concerned with the generality of the technique. Thus he extended the product monad to a monad constructor that took an existing monad and added the product monad. Gurr was not satisfied with this approach for several reasons. One was that embedding the cost directly into the semantics in the form of a product limited the type of categories to which the methodology could be applied. Second, Gurr wanted to be able to compare complexity of different languages based on different categories, which was more difficult when the actual costs are build into the structure of that category. Third, he had difficulty with this technique and higher-order functions, although that was primarily because he only considered external costs even though the calculus itself potentially included internal costs. Lastly, he wanted to be able to decide and reason about non-exact complexity and needed a more flexible structure in order to do so.

To solve his problems Gurr separated the category used to describe the language ($\mathcal{C}$) and the category used to described the complexity ($\mathcal{D}$) with a functor U from $\mathcal{C}$ to $\mathcal{D}$. Thus for a meaning of a program $f : A \to B$ in $\mathcal{C}$ he has a cost function from UA to a monoid M of costs. With this structure he then created a system of measures so that values are assigned sizes and sets of costs are assigned a specific cost (such as the maximum) to derive a function from sizes to costs. The resulting function form sizes to costs, was not compositional, so he extended the system by adding equivalences to the original cost functions instead of equivalence to the input and output of the cost functions.

The second system succeeds in its generality, but separating the cost functions from the value functions makes it difficult to extend this to accommodate internal cost, as required with higher order or lazy types. Gurr's first system is very similar to our system, except that we generalized it differently to remove the explicit use of costs in the system (instead in our framework the costs are mapped into morphisms that add cost). Also, Gurr did not have a system for explicitly adding costs except through the use of primitive functions. Lastly, our system maintained internal costs explicitly, making it more straightforward to handle higher-order and lazy types. Our system, however, is not as directly general, as Gurr examined systems for adding cost to general categorical semantics instead of a specific semantics as we did (although we included guidelines for adding cost to other semantics, just not a formal system).

6.1.4 Other work

There has been other research on semantics with cost. Sands' work was based on earlier work by Bjerner ([5]), Bjerner and Holmström ([6]), and Wadler ([45]). These authors explored the process of determining the time of a lazy functional program based on how much of the result is actually needed. All of their work was in first-order lazy languages, although none of them explored the effect of memoization.

There has also been work done in automatic analysis. In particular, there are several papers ([50], [21], [22], [36], [9]) that examine approximate costs of a function relative to its input without necessarily running the program on its input. Sands' system is also designed for automatic analysis, although he did not automate it in his thesis. His system also evaluates cost exactly, and while a cost function does calculate the value, it must calculate the value of anything needed in a branch or as an argument to another function. Some other systems, such as [36] and [9], work on approximating the input in such a way as to calculate cost for a wide variety of inputs

A more recent paper, [16] uses the cost monad to analyze a functional language that uses arrays. It uses a limited form of functional language to develop cost analysis based on the size and dimensions of the array and independent of the content of the arrays.

6.2 Future Work

6.2.1 Direct application

The technique developed in this dissertation for computing costs can be used directly in future projects. One particular use is in analyzing the complexity of other programs. The intensional semantics is primarily useful for programs, such as `accept`, that use higher order types in non-trivial ways, or for programs that manipulate lazy data structures but do not depend on memoization, such as the stream transformation programs found in [14] or the `tabulate` program in section 4.5.4.

Another possible use may be formally proving the correctness as well as the improvement in speed of certain types of local optimizations. Again this technique would be particular useful in the case where we have data types with internal costs.

$$\frac{e_1 \longrightarrow e_1'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \longrightarrow \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3}$$

$$\texttt{if true then } e_2 \texttt{ else } e_3 \longrightarrow e_2$$

$$\texttt{if false then } e_2 \texttt{ else } e_3 \longrightarrow e_3$$

$$\frac{e_1 \longrightarrow e_1'}{e_1(e_2) \longrightarrow e_1'(e_2)} \qquad \frac{e_2 \longrightarrow e_2'}{v(e_2) \longrightarrow v(e_2')}$$

$$(\texttt{lam } x.e)(v_2) \longrightarrow [v_2/x]e \qquad \texttt{rec } x.e \longrightarrow [\texttt{rec } x.e/x]e$$

Figure 6.1: A single-step call-by-value operational semantics

6.2.2 Comparisons with other operational semantics

Our choice of operational semantics is based on natural (big-step) transition rules using substitution because it is highly abstract yet concrete enough to allow costs based on counting the number of high-level operations. It is worthwhile, however, to see if we can derive sound denotational semantics relative to other types of operational semantics. For example, can we describe the number of steps in a single-step operational semantics? How does our semantics compare to an environment-based operational semantics instead of a substitution-based one? Can we describe costs associated with abstract machines such as the SECD machine?

Preliminary investigation indicates that we can use cost structures to derive a denotational semantics that accurately predicts the number of steps in a single-step operational semantics; for example the call-by-value denotational semantics defined in Chapter 3 is sound relative to the single step semantics shown in Figure 6.1 when we replace all costs (t_{rec}, etc.) with the integer 1. It also seems that an environment-based operational semantics might be a better basis for an intensional denotational semantics than the substitution-based semantics used in this dissertation. This is because the substitution-based semantics by nature duplicates code. When the duplicated code represented a value, it was therefore required that that code have 0 cost. This made it more difficult to separate constructors, which create values but in practice may spend time doing so, from the resulting values itself. More critically, we were required to set the cost of applying an abstraction to be a constant, independent of the abstraction itself. The only reason we were required to do so is that otherwise the Substitution Lemma fails to hold, thus leading to an unsound denotational semantics. With an environment-based semantics, we no longer have a need for the Substitution Lemma, and from preliminary examination it is likely that it is possible to define the cost of applying an abstraction as a function of the abstraction itself (for example, allowing the cost to be proportional to the number of free variables in the abstraction), thus allowing greater flexibility.

For abstract machines the comparison is more complicated, not only because such machines are further separated from the language itself, but also because there is more than one possible "cost" associated with such a machine. For example, with the SECD machine, one could count the number of state transitions or the number of stack operations. It would be useful to find out which operations can be described compositionally by the intensional semantics.

6.2.3 Parallelism

Another area for future exploration is the examination of costs associated with parallel computation. It is unlikely that our technique could be easily adapted to parallel settings that involve communication between processes, but a simple case of parallelism, where independent expressions are evaluated in parallel, should be describable using our techniques. For example, we could try to define compositional semantics for some of the operational semantics in Greiner's thesis ([10]). This would require, of course, a change in the cost structure because the set of costs would have to include two operations (one for combining sequential computation and one for combining parallel computation). This may turn out to be sufficient to model the case of parallel computation, but this has not been proven formally.

6.2.4 Call-by-need semantics

The call-by-name semantics is limited because it cannot always accurately describe situations in lazy languages that where an argument is evaluated either once, or not all, at different locations in the code and depending on external conditions. For example, given the function $\mathtt{lam}\, x.x{::}x{::}\mathtt{nil}$, it is not possible in the call-by-name strategy to have the expression bound to x evaluated both only when needed and no more than once.

Therefore another topic for future exploration is to derive an intensional semantics that can model call-by-need evaluation order. The difficulty encountered when describing call-by-need evaluation is that call-by-need evaluation inevitably requires some form of "state" which changes whenever a variable is evaluated. Therefore any denotational semantics that describes the costs associated with a call-by-need evaluation strategy must be able to handle state changes as well.

There are two standard ways in which state might be added within a category theoretic framework such as ours. The first uses the state monad $SX = X \times [S \Rightarrow S \times X]$, where the object S represents some form of state. In our case as we are keeping track of costs not values we would use a variant monad $SX = X \times [S \Rightarrow S \times T]$, with T being the object of costs. This could be then implemented by using a cost structure based on S with some extra functionality added to handle adding and updating the actual state.

An even more elegant method for adding state in category theory involves the use of a functor category. A functor category $\mathcal{C}^{\mathcal{W}}$ has functors from $\mathcal{W}$ to $\mathcal{C}$ as objects and natural transformations as morphisms. When used to model state transformations, $\mathcal{W}$ represents a category describing "possible worlds," which are loosely related to the set of known identifiers. Semantics involving functor categories has primarily been used to model Algol-like languages ([43], [34], [30]), although it has also been used to model references in ML-like languages ([42]).

With functor categories, we can theoretically define a cost structure in the category $\mathcal{C}^{\mathcal{W}}$. In addition, we need the ability to evaluate the state of an identifier (and take differing actions based on the outcome), and to add or remove new identifiers. With these extra structures, we can then define a semantics that describes a call-by-need semantics, using the new cost structure in place of the old, and with some possible changes to accommodate new identifiers and the evaluation of old ones.

Using functor categories is very elegant, even though calculating examples is necessarily more complex due to the need to manage state. Using a preliminary example (the details of which are beyond the scope of this chapter) we were able to show that, for example, the cost of applying `length` to a list is the same as for the call-by-value semantics plus the cost of evaluating the delayed costs stored within the tails of the list (the elements of the list are not evaluated).

Unfortunately, at this time we do not know if any cost structures using either the state functor or functor categories exist. Unlike the functor category semantics used in Algol and Forsythe, the values being stored as states are not basic types, but any allowable type. Thus the definition of any cost structure is circular making it very difficult to show that a solution is feasible, as well as not allowing any standard induction to prove properties. What makes this different in the call-by-need case is that a type is being used both covariantly and contravariantly, thus limiting the use of induction.

It may be possible to find a valid cost structure for this semantics, or it may be necessary to redesign the semantics. Currently this semantics is based on the operational semantics from [40]. It is possible that basing the denotational semantics on other operational semantics, particularly those based on graph reduction (such as in [**?**]), may provide insights that avoid the current problems.

6.2.5 Space complexity

Lastly, we would find it useful to be able to model space as well as time complexity. In this case the first step is to define an operational semantics that can describe costs related to space. In particular, any semantics using straight substitution would be ill suited for examination of space, as it would be difficult to avoid unnecessary duplication. Both Greiner's thesis ([10]) and research into compiler correctness (such as [7]), however, has generated several abstract methods for handling memory, many of which can then be used or adapted to create an operational model of memory usage. We could then see if our technique for calculation time-related costs can be adapted to calculating space related costs.

Appendix A

Adequacy of the call-by-value extensional semantics

The adequacy proofs for both the call-by-name and call-by-value semantics are taken mostly from [11], adjusted for the use of category theory and the separation of constants.

A.1 Adequacy of the call-by-value

In order to prove Lemma A.1.4 we need three additional lemmas. The first, Lemma A.1.1 simply makes proving adequacy easier. The second, Lemma A.1.2, assures us that recursion will work properly. The third, Lemma A.1.3, generalizes the assumption made about constants.

Lemma A.1.1 *For all morphisms $z : \mathbf{1} \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ and all closed expressions e_1, e_2 of type τ, if $z \precsim_\tau^{\mathrm{V}} e_1$ and if whenever there exists a value v such that $e_1 \Rightarrow_{\mathrm{v}} v$ then $e_2 \Rightarrow_{\mathrm{v}} v$, then $z \precsim_\tau^{\mathrm{V}} e_2$.*

Proof. Suppose that $z \precsim_\tau^{\mathrm{V}} e_1$. If $z = \bot$ then $z \precsim_\tau^{\mathrm{V}} e_2$. Otherwise there exists a value v such that $e_1 \Rightarrow_{\mathrm{v}} v$ and a morphism $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ such that $z = \mathsf{up} \circ y$ and $y \precsim_\tau^{\mathrm{V}*} v$. However, $e_2 \Rightarrow_{\mathrm{v}} v$ as well, so $z \precsim_\tau^{\mathrm{V}} e_2$. □

Definition A.1.1 A set X of morphisms from A to B is *inclusive* if the following holds:

1. If $f : A \to B$ is in X then for any $f' : A \to B$ such that $f' \leq f$, f' is in X.

2. If $f_1 \leq f_2 \leq \ldots$ are all morphisms from $A \to B$ such that each f_k is in X, then $\bigsqcup_{k=1}^{\infty} f_k$ is also in X.

Lemma A.1.2 *For all types τ and all closed expressions e of type τ, the set of morphisms $z : \mathbf{1} \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ for which $z \precsim_\tau^{\mathrm{V}} e$ is inclusive.*

Proof. To prove #1, if $z' = \bot$ then we immediately know that $z \precsim_\tau^{\mathrm{V}} e$, therefore suppose that $z' \neq \bot$. Then $z \neq \bot$, so there exists a value v such that $e \Rightarrow_{\mathrm{v}} v$ and a morphism $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ such that $z = \mathsf{up} \circ y$ and $y \precsim_\tau^{\mathrm{V}*} v$. Because $z' \neq \bot$ there exists a morphism $y' : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ such that $z' = \mathsf{up} \circ y'$. What is still needed is proof that $y' \precsim_\tau^{\mathrm{V}*} v$, done by induction on the type of v:

Case $\tau = g$: Immediate

Case $\tau = \delta(\tau_1, \ldots, \tau_n)$**:**

Note that because $z' \leq z$, $y' \leq y$,

$$\mathsf{up} \circ F_\delta(!, \ldots, !) \circ y' \leq \mathsf{up} \circ F_\delta(!, \ldots, !) \circ y \leq LF_\delta(!, \ldots, !) \circ \mathcal{V}_{\mathrm{E}}[\![v : \delta(\tau_1, \ldots, \tau_n)]\!]$$

Now let $p \in P_i$ be a deconstructor. Because $y \precsim^{\mathrm{V}*}_{\delta(\tau_1,\ldots,\tau_n)} v$, $p \circ y \precsim^{\mathrm{V}}_{\tau_i} [v/x]e_p$, and $p \circ y' \leq p \circ y$, by the induction hypothesis $p \circ y' \precsim^{\mathrm{V}}_{\tau_i} [v/x]e_p$.

Therefore $y' \precsim^{\mathrm{V}*}_{\delta(\tau_1,\ldots,\tau_n)} v$.

Case $\tau = \tau_1 \to \tau_2$**:**

Let $y_1 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!]$ and v_1 be a closed value of type τ_1 such that $y_1 \precsim^{\mathrm{V}*}_{\tau_1} v_1$. Then

$$\mathsf{app} \circ \langle y', \mathsf{up} \circ y_1 \rangle \leq \mathsf{app} \circ \langle y, \mathsf{up} \circ y_1 \rangle$$

and $\mathsf{app} \circ \langle y, \mathsf{up} \circ y_1 \rangle \precsim^{\mathrm{V}}_{\tau_2} v(v_1)$ so by the induction hypothesis

$$\mathsf{app} \circ \langle y', \mathsf{up} \circ y_1 \rangle \precsim^{\mathrm{V}}_{\tau_2} v(v_1)$$

Thus $y' \precsim^{\mathrm{V}}_{\tau_1 \to \tau_2} v$.

To prove #2, if, for all $k \geq 1$, $z_k = \bot$ then $\bigsqcup_{k=1}^{\infty} z_k = \bot$ so we immediately know that $\bigsqcup_{k=1}^{\infty} z_k \precsim^{\mathrm{V}}_{\tau} v$. Otherwise for some N, $z_N \neq \bot$, and thus there exists a value v such that $e \Rightarrow_{\mathrm{v}} v$ and there exists a morphism y_N such that $z_N = \mathsf{up} \circ y_N$ and $y_N \precsim^{\mathrm{V}*}_{\tau} v$. Because $\Rightarrow_{\mathrm{v}}$ is deterministic, it must hold that for all $k \geq N$, there exists a morphism $y_k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$ such that $z_k = \mathsf{up} \circ y_k$ and $y_k \precsim^{\mathrm{V}*}_{\tau} v$. Let $y = \bigsqcup_{k=N}^{\infty} y_k$. By the continuity of application, this means that

$$\bigsqcup_{k=1}^{\infty} z_k = \bigsqcup_{k=N}^{\infty} \mathsf{up} \circ y_k = \mathsf{up} \circ \bigsqcup_{k=N}^{\infty} y_k = \mathsf{up} \circ y$$

Therefore we must show that $y \precsim^{\mathrm{V}*}_{\tau} v$. The rest of the proof follows by induction on the structure of τ:

Case $\tau = g$**:**

The lemma follows directly from the fact that the set of morphisms from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$ is a complete partial order.

Case $\tau = \delta(\tau_1, \ldots, \tau_n)$**:**

First note that for each $k \geq 1$, $\mathsf{up} \circ F_\delta(!, \ldots, !) \circ y_k \leq LF_\delta(!, \ldots, !) \circ \mathcal{V}_{\mathrm{E}}[\![v : \delta(\tau_1, \ldots, \tau_n)]\!]$. Thus by the definition of least upper bounds,

$$\begin{aligned}
\mathsf{up} \circ F_\delta(!, \ldots, !) \circ y &= \mathsf{up} \circ F_\delta(!, \ldots, !) \circ \bigsqcup_{k=N}^{\infty} y_k \\
&= \bigsqcup_{k=N}^{\infty} \mathsf{up} \circ F_\delta(!, \ldots, !) \circ y_k \\
&\leq LF_\delta(!, \ldots, !) \circ \mathcal{V}_{\mathrm{E}}[\![v : \delta(\tau_1, \ldots, \tau_n)]\!]
\end{aligned}$$

Now let $p \in P_i$ be a deconstructor. Then for each $k \geq N$, $p \circ y_k \precsim^{\mathrm{V}}_{\tau_i} [v/x]e_p$, therefore by the induction hypothesis

$$\begin{aligned} p \circ y &= p \circ \bigsqcup_{k=N}^{\infty} y_k \\ &= \bigsqcup_{k=N}^{\infty} (p \circ y_k) \\ &\precsim^{\mathrm{V}}_{\tau_i} [v/x]e_p \end{aligned}$$

Thus $y \precsim^{\mathrm{V}^*}_{\delta(\tau_1,\ldots,\tau_n)} v$.

Case $\tau = \tau_1 \to \tau_2$:

Let $y_1 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_1]\!]$ and let v_1 be a closed value of type τ_1 such that $y_1 \precsim^{\mathrm{V}^*}_{\tau_1} v_1$. Because each $y_k \precsim^{\mathrm{V}}_{\tau_1\to\tau_2} v$, for all $k \geq N$, $\mathsf{app} \circ \langle y_k, \mathsf{up} \circ y_1\rangle \precsim^{\mathrm{V}}_{\tau_2} v(v_1)$. Therefore, by the induction hypothesis,

$$\begin{aligned} \mathsf{app} \circ \mathsf{smash} \circ \langle y, y_1\rangle &= \mathsf{app} \circ \langle \textstyle\bigsqcup_{k=N}^{\infty} y_k, y_1\rangle \\ &= \bigsqcup_{k=N}^{\infty} \mathsf{app} \circ \langle y_k, y_1\rangle \\ &\precsim^{\mathrm{V}}_{\tau_2} v(v_1) \end{aligned}$$

Thus $y \precsim^{\mathrm{V}^*}_{\tau_1\to\tau_2} v$.

$\square$

Lemma A.1.3 *Suppose that f is adequate for a constant $c \in \mathbf{Const}_{\tau_1\to\ldots\to\tau_n\to\tau}$ of arity n. For some $1 \leq i \leq n$, and for each $1 \leq j \leq i$, let $y_j : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_j]\!]$ and let v_j be a closed value of type τ_j such that $y_j \precsim^{\mathrm{V}^*}_{\tau_j} v_j$. Then*

$$\mathsf{raise}_{\perp}^{(n-i)}(f) \circ \langle\rangle(y_1,\ldots,y_i) \precsim^{\mathrm{V}}_{\tau_{i+1}\to\ldots\to\tau_n\ldots\tau} cv_1\ldots v_i$$

Proof. By induction on $n-i$. If $n = i$ then, by the assumption that f is adequate for c,

$$\begin{aligned} \mathsf{raise}_{\perp}^{(n-i)}(f) \circ \langle\rangle(y_1,\ldots,y_i) &= f \circ \langle\rangle(y_1,\ldots,y_n) \\ &\precsim^{\mathrm{V}}_{\tau} cv_1\ldots v_n \end{aligned}$$

If $n > i$, then $cv_1\ldots v_i$ has type $\tau_{i+1} \to \ldots \to \tau_n \to \ldots \to \tau$ so let $y' : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_{i+1}]\!]$ and let v' be a closed value of type τ_{i+1} such that $y' \precsim^{\mathrm{V}^*}_{\tau_{i+1}} v'$. Because $i < n$, $\mathsf{raise}_{\perp}^{(n-i)}(f) = \mathsf{up} \circ \mathsf{curry}(\mathsf{raise}_{\perp}^{(n-i-1)}(f))$ so all we need to do is show that

$$\mathsf{app} \circ \langle \mathsf{curry}(\mathsf{raise}_{\perp}^{(n-i-1)}(f)) \circ \langle\rangle(y_1,\ldots,y_i), y'\rangle \precsim^{\mathrm{V}}_{\tau_{i+2}\to\ldots\to\tau_n\to\tau} cv_1\ldots v_i v'$$

Let $\tau_0 = \tau_{i+2} \to \ldots \to \tau_n \to \tau$. Then by the induction hypothesis

$$\begin{aligned} \mathsf{app} \circ \langle \mathsf{curry}(\mathsf{raise}_{\perp}^{(n-i-1)}(f)) \circ \langle\rangle(y_1,\ldots,y_i), y'\rangle &= \mathsf{raise}_{\perp}^{(n-i-1)}(f) \circ \langle\rangle(y_1,\ldots,y_i,y') \\ &\precsim^{\mathrm{V}}_{\tau_0} cv_1\ldots v_i v' \end{aligned}$$

Thus
$$\mathsf{raise}_{\perp}^{(n-1)}(f) \circ \langle\rangle(y_1, \ldots, y_i) \precsim^{\mathrm{V}}_{\tau_{i+1} \to \tau_0} cv_1 \ldots v_i$$

□

Lemma A.1.4 *Suppose that $\Gamma \vdash e : \tau$, where $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$. Furthermore suppose that for $1 \leq i \leq n$, there exist $z_i : \mathbf{1} \to L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_i]\!]$ and closed expressions e_i of type τ_i such that $z_i \precsim^{\mathrm{V}}_{\tau_i} e_i$. Then*
$$\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!] \circ \langle\rangle(z_1, \ldots, z_n) \precsim^{\mathrm{V}}_{\tau} [e_1/x_1, \ldots, e_n/x_n]e$$

Proof. Let $[s]$ represent the substitution $[e_1/x_1, \ldots, e_n/x_n]$ and let $p_0 = \langle\rangle(f_1, \ldots, f_n)$.

If $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!] \circ p_0$ is $\perp$ then the result is immediate, so assume that $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!] \circ p_0 \neq \perp$. The rest follows by induction on the structure of e.

Case $e = x_i$:
$$\begin{array}{rcl}
\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash x_i : \tau_i]\!] \circ p_0 & = & \pi^n_i \circ \langle\rangle(z_1, \ldots, z_n) \\
& = & z_i \\
& \precsim^{\mathrm{V}}_{\tau_i} & e_i \\
& \equiv & [s]x_i
\end{array}$$

Case $e = c$:

By the assumption that $\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![-]\!]$ is adequate and by Lemma A.1.3,
$$\begin{array}{rcl}
\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash c : \tau]\!] \circ p_0 & = & \mathsf{raise}_{\perp}^{(\mathrm{ar}(c))}(\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c : \tau]\!]) \circ !_{\mathcal{T}^{\mathrm{V}}_{\perp}[\![\Gamma]\!]} \circ p_0 \\
& = & \mathsf{raise}_{\perp}^{(\mathrm{ar}(c))}(\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c : \tau]\!]) \circ \langle\rangle() \\
& \precsim^{\mathrm{V}}_{\tau} & c \\
& \equiv & [s]c
\end{array}$$

Case $e = \mathtt{lam}\ x.e'$, $\tau = \tau'' \to \tau'$:

Let $y'' : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau'']\!]$ and let v'' be a closed value of type τ'' such that $y'' \precsim^{\mathrm{V}*}_{\tau''} v''$. Let $y = \mathsf{curry}(\mathcal{V}_{\mathrm{E}}[\![\Gamma, x : \tau'' \vdash e' : \tau']\!] \circ (\mathsf{id} \times \mathsf{up})) \circ p_0$. Then $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!] \circ p_0 = \mathsf{up} \circ y$. Let $z : \mathbf{1} \to L\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau']\!]$ be the morphism $z = \mathsf{app} \circ \langle y, y'' \rangle$. Then $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!] \circ p_0 \precsim^{\mathrm{V}}_{\tau} [s]e$ if $z \precsim^{\mathrm{V}}_{\tau'} ([s]e)v''$. By the induction hypothesis,
$$\begin{array}{rcl}
z & = & \mathsf{app} \circ \langle y, y' \rangle \\
& = & \mathsf{app} \circ \mathsf{smash} \circ \langle \mathsf{up} \circ y, \mathsf{up} \circ y' \rangle \\
& = & \mathsf{app}^{\perp} \circ \mathsf{smash} \circ \langle \mathsf{up} \circ \mathsf{curry}(\mathcal{V}_{\mathrm{E}}[\![\Gamma, x : \tau'' \vdash e' : \tau']\!] \circ (\mathsf{id} \times \mathsf{up})) \circ p_0, \mathsf{up} \circ y'' \rangle \\
& = & \mathcal{V}_{\mathrm{E}}[\![\Gamma, x : \tau'' \vdash e' : \tau']\!] \circ (\mathsf{id} \times \mathsf{up}) \circ \langle p_0, y'' \rangle \\
& = & \mathcal{V}_{\mathrm{E}}[\![\Gamma, x : \tau'' \vdash e' : \tau']\!] \circ \langle p_0, \mathsf{up} \circ y'' \rangle \\
& \precsim^{\mathrm{V}}_{\tau'} & [s, v''/x]e' \\
& \equiv & [v''/x]([s]e')
\end{array}$$

Suppose that there exists a value v such that $[v''/x]([s]e') \Rightarrow_{\mathrm{v}} v$. Then, by the operational rules for application, $(\mathtt{lam}\ x.[s]e')(v'') \Rightarrow_{\mathrm{v}} v$. Therefore by Lemma A.1.1
$$z \precsim^{\mathrm{V}}_{\tau'} (\mathtt{lam}\ x.[s]e')(v'') \equiv ([s]\mathtt{lam}\ x.e')(v'') \equiv [s]e(v'')$$
so $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!] \circ p_0 \precsim^{\mathrm{V}}_{\tau} [s]e$.

Case $e = e_1(e_2)$**:**

Because $\Gamma \vdash e_1(e_2) : \tau$, there exists a type τ' such that $\Gamma \vdash e_1 : \tau' \to \tau$ and $\Gamma \vdash e_2 : \tau'$. By the induction hypothesis $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!] \circ p_0 \precsim^{\mathrm{V}}_{\tau' \to \tau} [s]e_1$. If $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!] \circ p_0 = \bot$, then

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1(e_2) : \tau]\!] \circ p_0} \\ = & \mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!] \rangle \circ p_0 \\ = & \mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle \bot, \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!] \circ p_0 \rangle \\ = & \bot \\ \precsim^{\mathrm{V}}_{\tau} & [s]e_1(e_2) \end{array}$$

If $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau']\!] \circ p_0 = \bot$, then

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1(e_2) : \tau]\!] \circ p_0} \\ = & \mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!] \rangle \circ p_0 \\ = & \mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!] \circ p_0, \bot \rangle \\ = & \bot \\ \precsim^{\mathrm{V}}_{\tau} & [s]e_1(e_2) \end{array}$$

Therefore assume that $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!] \circ p_0 \neq \bot$ and $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau']\!] \circ p_0 \neq \bot$. Then there exist morphisms $y_1 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau' \to \tau]\!]$ and $y_2 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau']\!]$ and values v_1, v_2 such that $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!] \circ p_0 = \mathsf{up} \circ y_1$, $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau']\!] \circ p_0 = \mathsf{up} \circ y_2$, $[s]e_1 \Rightarrow_{\mathrm{v}} v_1$, $[s]e_2 \Rightarrow_{\mathrm{v}} v_2$, $y_1 \precsim^{\mathrm{V}^*}_{\tau' \to \tau} v_1$, and $y_2 \precsim^{\mathrm{V}^*}_{\tau'} v_2$. Therefore by the definition of $\precsim^{\mathrm{V}^*}$ for functional types,

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1(e_2) : \tau]\!] \circ p_0} \\ = & \mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!] \circ p_0, \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!] \circ p_0 \rangle \\ = & \mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle \mathsf{up} \circ y_1, \mathsf{up} \circ y_2 \rangle \\ = & \mathsf{app} \circ \langle y_1, y_2 \rangle \\ \precsim^{\mathrm{V}}_{\tau} & v_1(v_2) \end{array}$$

Suppose there exists a value v such that $v_1(v_2) \Rightarrow_{\mathrm{v}} v$. There are two possible derivations of $v_1(v_2) \Rightarrow_{\mathrm{v}} v$. The first, where v_1 is of the form $\texttt{lam}\ x.e'$, is the derivation

$$\frac{v_1 \Rightarrow_{\mathrm{v}} \texttt{lam}\ x.e' \quad v_2 \Rightarrow_{\mathrm{v}} v_2 \quad [v_2/x]e' \Rightarrow_{\mathrm{v}}{}' v}{v_1(v_2) \Rightarrow_{\mathrm{v}} v}$$

As $[s]e_1 \Rightarrow_{\mathrm{v}} v_1$ and $[s]e_2 \Rightarrow_{\mathrm{v}} v_2$ the same rule can be applied to derive that $([s]e_1)([s]e_2) \Rightarrow_{\mathrm{v}} v$ which means that $[s]e_1(e_2) \Rightarrow_{\mathrm{v}} v$.

Otherwise for some constant c, $i < \mathrm{ar}(c)$, and values $v'_1, \ldots, v'_i$, $v_1 = cv'_1 \ldots v'_i$, the derivation of $v_1(v_2) \Rightarrow_{\mathrm{v}} v$ is of the form

$$\frac{v_1 \Rightarrow_{\mathrm{v}} cv'_1 \ldots v'_i \quad v_2 \Rightarrow_{\mathrm{v}} v_2 \quad \mathrm{vapply}(v_1, v_2) \Rightarrow v}{v_1(v_2) \Rightarrow_{\mathrm{v}} v}$$

It is possible, once again, to derive that $[s]e_1(e_2) \Rightarrow_{\mathrm{v}} v$. Therefore by Lemma A.1.1,

$$\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1(e_2) : \tau]\!] \precsim^{\mathrm{V}}_{\tau} [s]e_1(e_2)$$

Case $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$:

Let $z = \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!] \circ p_0$. By the induction hypothesis $z \precsim^{\mathrm{V}}_{\mathbf{bool}} [s]e_1$. If $z = \bot$ then

$$\begin{array}{rl} \multicolumn{2}{l}{\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau]\!] \circ p_0} \\ = & \mathsf{cond}^{\bot} \circ \mathsf{smash} \circ (\mathsf{id} \times \mathsf{up}) \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \mathbf{bool}]\!], \\ & \quad \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_3 : \tau]\!]\rangle\rangle \circ p_0 \\ = & \mathsf{cond}^{\bot} \circ \mathsf{smash} \circ \langle \bot, \mathsf{up} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_3 : \tau]\!]\rangle \circ p_0\rangle \\ = & \bot \\ \precsim^{\mathrm{V}}_{\tau} & [s]\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \end{array}$$

Therefore suppose that $z \neq \bot$. Then there exists a morphism $y : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\mathbf{bool}]\!]$ and value v_1 such that $z = \mathsf{up} \circ y$, $[s]e_1 \Rightarrow_{\mathrm{v}} v_1$, and $y \precsim^{\mathrm{V}*}_{\mathbf{bool}} v_1$. Because **bool** is a ground type, $\mathsf{up} \circ v \leq \mathcal{V}_{\mathrm{E}}[\![v_1 : \mathbf{bool}]\!]$, so by the construction of the boolean object $\mathbf{B}$ either $y = \mathsf{tt}$ and $v_1 = \texttt{true}$, or $y = \mathsf{ff}$ and $v_1 = \texttt{false}$.

If $y = \mathsf{tt}$, then by the induction hypothesis,

$$\begin{array}{rl} \multicolumn{2}{l}{\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau]\!] \circ p_0} \\ = & \mathsf{cond}^{\bot} \circ \mathsf{smash} \circ \langle \mathsf{up} \circ \mathsf{tt}, \mathsf{up} \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!], \\ & \quad \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_3 : \tau]\!]\rangle \circ p_0\rangle \\ = & \mathsf{cond} \circ \langle \mathsf{tt}, \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_3 : \tau]\!]\rangle \circ p_0\rangle \\ = & \pi_1 \circ \langle \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!], \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_3 : \tau]\!]\rangle \circ p_0 \\ = & \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!] \circ p_0 \\ \precsim^{\mathrm{V}}_{\tau} & [s]e_2 \end{array}$$

Suppose there exists a value v such that $[s]e_2 \Rightarrow_{\mathrm{v}} v$. Then by the derivation

$$\frac{[s]e_1 \Rightarrow_{\mathrm{v}} \texttt{true} \quad [s]e_2 \Rightarrow_{\mathrm{v}} v}{\texttt{if}\ [s]e_1\ \texttt{then}\ [s]e_2\ \texttt{else}\ [s]e_3 \Rightarrow_{\mathrm{v}} v}$$

we know that by Lemma A.1.1

$$\begin{array}{rcl} \mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau]\!] \circ p_0 & \precsim^{\mathrm{V}}_{\tau} & \texttt{if}\ [s]e_1\ \texttt{then}\ [s]e_2\ \texttt{else}\ [s]e_3 \\ & \equiv & [s]\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \end{array}$$

Similarly, if $y = \mathsf{ff}$, then $\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau]\!] \circ p_0 \precsim^{\mathrm{V}}_{\tau} [s]e_3$, so by similar arguments and the derivation

$$\frac{e_1 \Rightarrow_{\mathrm{v}} \texttt{false} \quad [s]e_3 \Rightarrow_{\mathrm{v}} v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow_{\mathrm{v}} v}$$

$\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau]\!] \circ p_0 \precsim^{\mathrm{V}}_{\tau} [s]\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$.

Case $e = \texttt{rec}\ x.e'$:

Let $z'_0 = \bot_{\mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]}$ and for $k > 0$ let

$$z'_k = \mathcal{V}_{\mathrm{E}}[\![\Gamma, x : \tau \vdash e' : \tau]\!] \circ \langle p_0, z'_{k-1}\rangle = \mathcal{V}_{\mathrm{E}}[\![\Gamma, x : \tau \vdash e' : \tau]\!] \circ \langle\rangle(z_1, \ldots, z_n, z'_{k-1})$$

By definition $z'_0 \precsim^{\mathrm{V}}_{\tau} [s]\texttt{rec}\ x.e'$. Suppose that $z'_{k-1} \precsim^{\mathrm{V}}_{\tau} [s]\texttt{rec}\ x.e'$. Then by induction hypothesis on e,

$$\begin{array}{rcl} z'_k & = & \mathcal{V}_{\mathrm{E}}[\![\Gamma, x:\tau \vdash e':\tau]\!] \circ \langle\rangle(z_1,\ldots,z_n,z'_{k-1}) \\ & \precsim^{\mathrm{V}}_{\tau} & [s,[s]\texttt{rec}\ x.e'/x]e' \\ & \equiv & [[s]\texttt{rec}\ x.e'/x][s]e' \end{array}$$

Suppose that there exists a v such that $[[s]\texttt{rec}\ x.e'/x][s]e' \Rightarrow_{\mathrm{v}} v$. Then by the derivation

$$\frac{[[s]\texttt{rec}\ x.e'/x]e' \Rightarrow_{\mathrm{v}} v}{[s]\texttt{rec}\ x.e' \Rightarrow_{\mathrm{v}} v}$$

and Lemma A.1.1, $z'_k \precsim^{\mathrm{V}}_{\tau} [s]\texttt{rec}\ x.e'$. Thus by induction on k, for all $k \geq 0$, $z'_k \precsim^{\mathrm{V}}_{\tau} [s]\texttt{rec}\ x.e'$, so by the definition of fixp,

$$\begin{array}{rl} \multicolumn{2}{l}{\mathcal{V}_{\mathrm{E}}[\![\Gamma \vdash \texttt{rec}\ x.e:\tau]\!] \circ p_0} \\ = & \mathsf{fixp}(\mathcal{V}_{\mathrm{E}}[\![\Gamma, x:\tau \vdash e':\tau]\!]) \circ p_0 \\ = & \mathsf{fixp}(\mathcal{V}_{\mathrm{E}}[\![\Gamma, x:\tau \vdash e':\tau]\!] \circ (p_0 \times \mathsf{id})) \\ = & \bigsqcup_{k=0}^{\infty} f'_k \\ \precsim^{\mathrm{V}}_{\tau} & [s]\texttt{rec}\ x.e' \end{array}$$

□

A.2 Call-by-value adequacy of the FL constants

The proofs that various constants are adequate depends upon Lemma 3.4.8, which states that the meaning of a value factors through η. While that lemma applies to the intensional semantics, by applying E throughout it is clear that the extensional meaning of a value factors through up. That lemma did not depend on the adequacy of the extensional semantics so it is safe to apply it in this section.

The proof that constants on ground types are adequate applies not just for FL, but for any language built on top of the base language (that is for any set of constants).

Theorem A.2.1 *Suppose that c is a constant of arity n and type $\tau_1 \to \ldots \to \tau_n \to \tau$ where each τ_i and τ are ground types. Then $\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c]\!]$ is adequate for c.*

Proof. For $1 \leq i \leq n$, let $y_i : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_i]\!]$ and let v_i be a closed value of type τ_i such that $y_i \precsim^{\mathrm{V}^*}_{\tau} v_i$. Because each τ_i is a ground type, for all i, $\mathsf{up} \circ y_i \leq \mathcal{V}_{\mathrm{E}}[\![v_i:\tau_i]\!]$. Furthermore, because each v_i is a value, there exists a $y'_i : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau_i]\!]$ such that $\mathcal{V}_{\mathrm{E}}[\![v_i:\tau_i]\!] = \mathsf{up} \circ y'_i$. Therefore

$$\begin{array}{rcl} \mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c]\!] \circ \langle\rangle(y_1,\ldots,y_n) & \leq & \mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c]\!] \circ \langle\rangle(y'_1,\ldots,y'_n) \\ & = & \mathcal{V}_{\mathrm{E}}[\![cv_1\ldots v_n]\!] \end{array}$$

so, by the definition of $\precsim^{\mathrm{V}^*}$ for ground types, $\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![c]\!] \circ \langle\rangle(y_1,\ldots,y_n) \precsim^{\mathrm{V}}_{\tau} \mathcal{V}_{\mathrm{E}}[\![cv_1\ldots v_n]\!]$, proving that c is adequate. □

The proof that the product, sum, and list constants are adequate is separated into two parts: first, we define a more intuitive (for that type) definition of $\precsim^{\mathrm{V}^*}_{\tau}$ and show that the primary definition is equivalent. Second, we use the new definition to show that the constants are adequate. While it is not strictly necessary to separate the proofs, seeing that the standard definition matches a more intuitive definition is by itself useful.

A.2.1 Products

For products, there are two deconstructors, $p_{\times_1} = \mathsf{up} \circ \pi_1$ and $p_{\times_2} = \mathsf{up} \circ \pi_2$, with related expressions $e_{\times_1} = \mathtt{fst}(x)$ and $e_{\times_2} = \mathtt{snd}(x)$.

Lemma A.2.2 *Let $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1 \times \tau_2]\!]$ and let v be a closed value of type $\tau_1 \times \tau_2$. Then v'$y \precsim^{\mathrm{V}*}_{\tau_1 \times \tau_2} v$ if and only if there exist morphisms $y_1 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1]\!]$, $y_2 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_2]\!]$ such that $y = \langle y_1, y_2 \rangle$, $\mathsf{up} \circ y_1 \precsim^{\mathrm{V}}_{\tau_1} \mathtt{fst}(v_1)$, and $\mathsf{up} \circ y_2 \precsim^{\mathrm{V}}_{\tau_2} \mathtt{snd}(v_2)$.*

Proof. Suppose that $y \precsim^{\mathrm{V}*}_{\tau_1 + \tau_2} v$. Then by the uniqueness of products, if we let $y_1 = \pi_1 \circ y$ and $y_2 = \pi_2 \circ y$, $y = \langle y_1, y_2 \rangle$. Furthermore,

$$\mathsf{up} \circ y_1 = \mathsf{up} \circ \pi_1 \circ y = \mathsf{up} \circ p_{\times_1} \precsim^{\mathrm{V}}_{\tau_1} \mathtt{fst}(v)$$

Similarly, $\mathsf{up} \circ y_2 \precsim^{\mathrm{V}}_{\tau_2} \mathtt{snd}(v)$. Therefore the first part of the lemma holds.

To prove the equivalence in the other direction, suppose there exists morphisms $y_1 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1]\!]$ and $y_2 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_2]\!]$ such that $y = \langle y_1, y_2 \rangle$, $\mathsf{up} \circ y_1 \precsim^{\mathrm{V}}_{\tau_1} \mathtt{fst}(v)$, and $\mathsf{up} \circ y_2 \precsim^{\mathrm{V}}_{\tau_2} \mathtt{snd}(v)$.

Because v is a value, its meaning factors through up. Therefore there exists a morphism $y' : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1 \times \tau_2]\!]$ such that $\mathcal{V}_{\mathrm{E}}[\![v : \tau_1 \times \tau_2]\!] = \mathsf{up} \circ y'$. This means that

$$L(! \times !) \circ \mathcal{V}_{\mathrm{E}}[\![<v_1, v_2>]\!] = L(! \times !) \circ \mathsf{up} \circ y' = \mathsf{up} \circ \langle !, ! \rangle$$

Because $\mathsf{up} \circ \langle !, ! \rangle$ is the top value in the predomain of morphisms from $\mathbf{1}$ to $L(\mathbf{1} \times \mathbf{1})$, it is thus automatically true that $\mathsf{up} \circ (! \times !) \circ y \leq L(! \times !) \circ \mathcal{V}_{\mathrm{E}}[\![<v_1, v_2>]\!]$. Furthermore,

$$\mathsf{up} \circ \pi_1 \circ y = \mathsf{up} \circ y_1 \precsim^{\mathrm{V}}_{\tau_1} \mathtt{fst}(v)$$

Similarly, $\mathsf{up} \circ \pi_2 \circ y \precsim^{\mathrm{V}}_{\tau_2} \mathtt{snd}(v)$. Therefore $y \precsim^{\mathrm{V}*}_{\tau_1 \times \tau_2} v$. □

With Lemma A.2.2 it is a simple process to show that the product constants are adequate. Let y be a morphism from $\mathbf{1}$ to $\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1 \times \tau_2]\!]$ and let v be a closed value of type $\tau_1 \times \tau_2$ such that $y \precsim^{\mathrm{V}*}_{\tau_1 \times \tau_2} v$. Then we know that there exist morphisms y_1 and y_2 such that $y = \langle y_1, y_2 \rangle$, $\mathsf{up} \circ y_1 \precsim^{\mathrm{V}}_{\tau_1} \mathtt{fst}(v)$, and $\mathsf{up} \circ y_2 \precsim^{\mathrm{V}}_{\tau_2} \mathtt{snd}(v)$. Therefore

$$\begin{array}{rcl} \mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{fst}]\!] \circ \langle\rangle(y) & = & \mathsf{up} \circ \pi_1 \circ \langle y_1, y_2 \rangle \\ & = & \mathsf{up} \circ y_1 \\ & \precsim^{\mathrm{V}}_{\tau_1} & \mathtt{fst}(v) \end{array}$$

Similarly $\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{snd}]\!] \circ \langle\rangle(f) \precsim^{\mathrm{V}}_{\tau_2} \mathtt{snd}(v)$. Therefore $\mathtt{fst}$ and $\mathtt{snd}$ are adequate.

For $\mathtt{pair}$, let $y_1 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1]\!]$ and $y_2 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_2]\!]$. Let v_1 be a closed value of type τ_1, and let v_2 be a closed value of type τ_2 such that $y_1 \precsim^{\mathrm{V}*}_{\tau_1} v_1$ and $y_2 \precsim^{\mathrm{V}*}_{\tau_2} v_2$. Then, because $\mathtt{fst}(<v_1, v_2>) \Rightarrow_{\mathrm{v}} v_1$ and $\mathtt{snd}(<v_1, v_2>) \Rightarrow_{\mathrm{v}} v_2$, we know that $y_1 \precsim^{\mathrm{V}*}_{\tau_1} \mathtt{fst}(<v_1, v_2>)$ and $y_2 \precsim^{\mathrm{V}*}_{\tau_2} \mathtt{snd}(<v_1, v_2>)$. Thus

$$\langle y_1, y_2 \rangle \precsim^{\mathrm{V}*}_{\tau_1 + \tau_2} <v_1, v_2>$$

which means that $\mathsf{up} \circ \langle y_1, y_2 \rangle \precsim^{\mathrm{V}}_{\tau_1 + \tau_2} <v_1, v_2>$. Therefore

$$\begin{array}{rcl} \mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{pair}]\!] \circ \langle\rangle(f_1, f_2) & = & \mathsf{up} \circ (\pi_2 \times \mathsf{id}) \circ \langle\rangle(y_1, y_2) \\ & = & \mathsf{up} \circ \langle y_1, y_2 \rangle \\ & \precsim^{\mathrm{V}}_{\tau_1 + \tau_2} & <v_1, v_2> \end{array}$$

so $\mathtt{pair}$ is adequate.

A.2.2 Adequacy of Sums

In section 2.7.5 we showed that for any $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1 + \tau_2]\!]$, $y \precsim_{\tau_1+\tau_2}^{\mathrm{V}*} v$ if and only if one of the following properties hold:

- For some $y' : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1]\!]$ and closed term v' of type τ_1, $y = \mathsf{inl} \circ y'$, $v = \mathtt{inl}(v')$ and $y' \precsim_{\tau_1}^{\mathrm{V}*} v'$, or
- For some $y' : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_2]\!]$ and closed term v' of type τ_2, $y = \mathsf{inr} \circ y'$, $v = \mathtt{inr}(v')$ and $y' \precsim_{\tau_2}^{\mathrm{V}*} v'$

We use this lemma to show that the sum constants are adequate. For `inl`, let v be a closed value of type τ_1 and let $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1]\!]$ such that $y \precsim_{\tau_1}^{\mathrm{V}*} v$. Then we know that

$$\mathsf{inl} \circ y \precsim_{\tau_1+\tau_2}^{\mathrm{V}*} \mathtt{inl}(v)$$

However, $\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{inl}]\!] \circ \langle\rangle(y) = \mathsf{up} \circ \mathsf{inl} \circ y$, so $\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{inl}]\!] \circ \langle\rangle(y) \precsim_{\tau_1+\tau_2}^{\mathrm{V}} \mathtt{inl}(v)$ and thus `inl` is adequate. Similarly, we can show that `inr` is adequate.

For `case`, let v, v_1 and v_2 be closed values of types $\tau_1 + \tau_2$, $\tau_1 \to \tau$ and $\tau_2 \to \tau$ respectively. Also let $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1 + \tau_2]\!]$, $y_1 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1 \to \tau]\!]$, and $y_2 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_2 \to \tau]\!]$ such that $y \precsim_{\tau_1+\tau_2}^{\mathrm{V}*} v$, $y_1 \precsim_{\tau_1\to\tau}^{\mathrm{V}*} v_1$ and $y_2 \precsim_{\tau_2\to\tau}^{\mathrm{V}*} v_2$. Then `case` is adequate if

$$\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{case}]\!] \circ \langle\rangle(y, y_1, y_2) \precsim_{\tau}^{\mathrm{V}} \mathtt{case}\ v\ \mathtt{of}\ \mathtt{left} : v_1\ \mathtt{right} : v_2$$

Suppose that for some $y' : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau_1]\!]$, $y = \mathsf{inl} \circ y'$. Then as $y \precsim_{\tau_1+\tau_2}^{\mathrm{V}*} v$, there must exists a value v' of type τ_1 such that $v = \mathtt{inl}(v')$ and $y' \precsim_{\tau_1}^{\mathrm{V}*} v'$. Furthermore, because $y_1 \precsim_{\tau_1\to\tau}^{\mathrm{V}*} v_1$,

$$\mathsf{app} \circ \langle y_1, y' \rangle \precsim_{\tau}^{\mathrm{V}} v_1(v')$$

Thus

$$\begin{array}{rcl}
\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{case}]\!] \circ \langle\rangle(y, y_1, y_2) & = & \mathsf{case}(\mathsf{vleft}, \mathsf{vright}) \circ (\pi_2 \times \mathsf{id}) \circ \alpha^r \circ \langle\rangle(y, y_1, y_2) \\
& = & \mathsf{case}(\mathsf{vleft}, \mathsf{vright}) \circ \langle \mathsf{inl} \circ y', \langle y_1, y_2 \rangle\rangle \\
& = & \mathsf{vleft} \circ \langle y', \langle y_1, y_2 \rangle\rangle \\
& = & \mathsf{app} \circ \beta \circ (\mathsf{id} \times \pi_1) \circ \langle y', \langle y_1, y_2 \rangle\rangle \\
& = & \mathsf{app} \circ \langle y_1, y' \rangle \\
& \precsim_{\tau}^{\mathrm{V}} & v_1(v')
\end{array}$$

Suppose there exists a value v'' such that $v_1(v') \Rightarrow_{\mathrm{v}} v''$. Then with the derivation

$$\frac{v_1(v') \Rightarrow_{\mathrm{v}} v''}{\mathrm{vapply}(\mathtt{case}(\mathtt{inl}(v'))(v_1), v_2) \Rightarrow v''}$$

we know that $\mathtt{case}\ \mathtt{inl}(v')\ \mathtt{of}\ \mathtt{left} : v_1\ \mathtt{right} : v_2 \Rightarrow_{\mathrm{v}} v''$ as well. Therefore by Lemma A.1.1

$$\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{case}]\!] \circ \langle\rangle(y, y_1, y_2) \precsim_{\tau}^{\mathrm{V}} \mathtt{case}\ v\ \mathtt{of}\ \mathtt{left} : v_1\ \mathtt{right} : v_2$$

A similar proof shows that if, for some morphism y', $y = \mathsf{inr} \circ y'$ then

$$\mathcal{C}^{\mathrm{V}}[\![\mathtt{case}]\!] \circ \langle\rangle(y, y_1, y_2) \precsim_{\tau}^{\mathrm{V}} \mathtt{case}\ v\ \mathtt{of}\ \mathtt{left} : v_1\ \mathtt{right} : v_2$$

Therefore `case` is adequate.

A.2.3 Adequacy of lists

We have a deconstructor for each potential item on a list, i.e., $p_{L_1} = \mathsf{hd}$ and, for $n > 1$, $p_{L_k} = {p_{L_k - 1}}^{\perp} \circ \mathsf{tl}$. The related expressions are $e_{L_k} = \mathtt{head}(\mathtt{tail}^{(k-1)}(x))$.

Because the deconstructors are less aligned with the list constants, proving adequacy is somewhat more complicated; therefore developing a more intuitive method for determining adequacy is particularly useful.

Lemma A.2.3 *Suppose that there exist closed values v_1 and v_2 with $\vdash v_1 : \tau$ and $\vdash v_2 : \mathbf{list}(\tau)$. Then, for all $k > 0$, $\mathtt{tail}^{(k)}(v_1{::}v_2) \Rightarrow_{\mathrm{v}} v$ if and only if $\mathtt{tail}^{(k-1)}(v_2) \Rightarrow_{\mathrm{v}} v$.*

Proof. Straightforward by induction on k. □

Lemma A.2.4 *Let $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{list}(\tau)]\!]$ and v be a closed value of type $\mathbf{list}(\tau)$. Then $y \precsim^{\mathrm{V}*}_{\mathbf{list}(\tau)} v$ if and only if one of the following holds:*

- *$y = \mathsf{nil}$ and $v = \mathtt{nil}$, or*
- *There exist morphisms $y_1 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$, $y_2 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{list}(\tau)]\!]$ and values $v_1 : \tau$, $v_2 : \mathbf{list}(\tau)$ such that $y = \mathsf{cons} \circ \langle y_1, y_2\rangle$, $v = v_1{::}v_2$, $y_1 \precsim^{\mathrm{V}*}_{\tau} v_1$, and $y_2 \precsim^{\mathrm{V}*}_{\mathbf{list}(\tau)} v_2$.*

Proof.

$\Rightarrow$: Suppose that $y \precsim^{\mathrm{V}*}_{\mathbf{list}(\tau)} v$.

Suppose that $y = \mathsf{nil}$. Then, because $\mathsf{up} \circ \mathsf{List}\,(!) \circ y \leq L(\mathsf{List}\,(!)) \circ \mathcal{V}_{\mathrm{E}}[\![v : \mathbf{list}(\tau)]\!]$, we know that

$$\mathsf{up} \circ \mathsf{List}\,(!) \circ \mathsf{nil} = L(\mathsf{List}\,(!)) \circ \mathsf{up} \circ \mathsf{nil} \leq L(\mathsf{List}\,(!)) \circ \mathcal{V}_{\mathrm{E}}[\![v : \mathbf{list}(\tau)]\!]$$

The only value of v which can satisfy the previous equation is $v = \mathtt{nil}$, thus proving the lemma.

If $y \neq \mathsf{nil}$, then there must exist morphisms $y_1 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ and $y_2 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{list}(\tau)]\!]$ such that $y = \mathsf{cons} \circ \langle y_1, y_2\rangle$. In that case the only way

$$\mathsf{up} \circ \mathsf{List}\,! \circ y = L(\mathsf{List}\,(!)) \circ \mathsf{up} \circ z \leq L(\mathsf{List}\,(!)) \circ \mathcal{V}_{\mathrm{E}}[\![v : \mathbf{list}(\tau)]\!]$$

can hold is if there exists values v_1 and v_2 such that $v = v_1{::}v_2$.

What remains is to prove that $y_1 \precsim^{\mathrm{V}*}_{\tau} v_1$ and $y_2 \precsim^{\mathrm{V}*}_{\mathbf{list}(\tau)} v_2$. For y_1, using the first projection we know that

$$\begin{aligned} \mathsf{up} \circ y_1 &= \mathsf{up} \circ \pi_1 \circ \langle y_1, y_2\rangle \\ &= \mathsf{hd} \circ \mathsf{cons} \circ \langle y_1, y_2\rangle \\ &= \mathsf{hd} \circ y \\ &\precsim^{\mathrm{V}}_{\tau} \mathtt{head}(v) \end{aligned}$$

Furthermore, with the following derivation

$$\frac{\mathtt{head} \Rightarrow_{\mathrm{v}} \mathtt{head} \quad \mathrm{vapply}(\mathtt{head}, v_1{::}v_2) \Rightarrow v_1}{\mathtt{head}(v_1{::}v_2) \Rightarrow_{\mathrm{v}} v_1}$$

we know that $\mathtt{head}(v) \Rightarrow_{\mathrm{v}} v_1$, so $y_1 \precsim^{\mathrm{V}*}_{\tau} v_1$.

For y_2, we know that

$$\begin{aligned}&\mathsf{up} \circ \mathsf{cons} \circ \langle !, \mathsf{List}\,(!) \circ y_2\rangle \\ &\quad = \quad \mathsf{up} \circ \mathsf{cons} \circ (! \times \mathsf{List}\,(!)) \circ \langle y_1, y_2\rangle \\ &\quad = \quad \mathsf{up} \circ \mathsf{List}\,(!) \circ \mathsf{cons} \circ \langle y_1, y_2\rangle \\ &\quad = \quad \mathsf{up} \circ \mathsf{List}\,(!) \circ y\end{aligned}$$

Furthermore, as v_1 and v_2 are values, we know that there exist morphism y'_1 and y'_2 such that $\mathcal{V}_{\mathrm{E}}[\![v_1 : \tau]\!] = \mathsf{up} \circ y'_1$ and $\mathcal{V}_{\mathrm{E}}[\![v_2 : \mathbf{list}\,(\tau)]\!] = \mathsf{up} \circ y'_2$. Thus

$$\begin{aligned}L(\mathsf{List}\,(!)) \circ \mathcal{V}_{\mathrm{E}}[\![v : \mathbf{list}\,(\tau)]\!] = &\quad L(\mathsf{List}\,(!)) \circ \mathsf{up} \circ \mathsf{cons} \circ \langle y'_1, y'_2\rangle \\ = &\quad \mathsf{up} \circ \mathsf{cons} \circ \langle !, \mathsf{List}\,(!) \circ y'_2\rangle\end{aligned}$$

Thus because the ordering on lists is pointwise, we know that $\mathsf{List}\,(!) \circ y_2 \leq \mathsf{List}\,(!) \circ y'_2$, i.e.,

$$\mathsf{up} \circ \mathsf{List}\,(!) \circ y_2 \leq \mathsf{up} \circ \mathsf{List}\,(!) \circ y'_2 = L(\mathsf{List}\,(!)) \circ \mathcal{V}_{\mathrm{E}}[\![v_2 : \mathbf{list}\,(\tau)]\!]$$

Also,

$$\begin{aligned}p_{L_k} \circ y_2 = &\quad {p_{L_k}}^{\perp} \circ \mathsf{up} \circ \pi_2 \circ \langle y_1, y_2\rangle \\ = &\quad {p_{L_k}}^{\perp} \circ \mathsf{tl} \circ \mathsf{cons} \circ \langle y_1, y_2\rangle \\ = &\quad p_{L_k+1} \circ \mathsf{cons} \circ \langle y_1, y_2\rangle \\ = &\quad p_{L_k+1} \circ y\end{aligned}$$

Therefore $p_{L_k} \circ y_2 =\precsim^{\mathrm{V}}_{\tau} [v/x]e_{L_{k+1}} \equiv \texttt{head}(\texttt{tail}^{(k)}(v_1{::}v_2))$. Suppose there exists a value v' such that $\texttt{head}(\texttt{tail}^{(k)}(v_1{::}v_2)) \Rightarrow_{\mathrm{v}} v'$. The only derivation possible for the evaluation is

$$\frac{\texttt{head} \overset{0}{\Rightarrow}_{\mathrm{v}} \texttt{head} \quad \texttt{tail}^{(k)}(v_1{::}v_2) \Rightarrow_{\mathrm{v}} v'{::}v'' \quad \mathrm{apply}(\texttt{head}, v'{::}v'') \Rightarrow v'}{\texttt{head}(\texttt{tail}^{(k)}(v_1{::}v_2)) \Rightarrow_{\mathrm{v}} v'}$$

However, by Lemma A.2.3 we know that if $\texttt{tail}^{(k)}(v_1{::}v_2) \Rightarrow_{\mathrm{v}} v'{::}v''$ then $\texttt{tail}^{(k-1)}(v_2) \Rightarrow_{\mathrm{v}} v'{::}v''$, which means that $\texttt{head}(\texttt{tail}^{(k-1)}(v_2)) \Rightarrow_{\mathrm{v}} v'$. Thus by Lemma A.1.1

$$p_{L_k} \circ y_2 \precsim^{\mathrm{V}}_{\tau} [v_2/x]e_{L_k} \equiv \texttt{head}(\texttt{tail}^{(k-1)}(v_2))$$

so $y_2 \precsim^{\mathrm{V}*}_{\mathbf{list}(\tau)} v_2$.

$\Leftarrow$: Suppose that one of the properties of the lemma holds.

If $y = \mathsf{nil}$ and $v = \texttt{nil}$, then $\mathsf{up} \circ y = \mathcal{V}_{\mathrm{E}}[\![\texttt{nil}]\!]$, so

$$\mathsf{up} \circ \mathsf{List}\,(!) \circ y = L(\mathsf{List}\,(!)) \circ \mathsf{up} \circ y = L(\mathsf{List}\,(!)) \circ \mathcal{V}_{\mathrm{E}}[\![\texttt{nil}]\!]$$

Furthermore,

$$p_{L_1} \circ y = \mathsf{hd} \circ \mathsf{nil} = \perp$$

and, for $k > 1$,

$$p_{L_k} \circ y = {p_{L_k-1}}^{\perp} \circ \mathsf{tl} \circ \mathsf{nil} = \perp$$

Therefore for all $k > 0$, $p_{L_k} \circ y = \perp \precsim^{\mathrm{V}}_{\tau} [v/x]e_{L_k}$. Thus $y \precsim^{\mathrm{V}*}_{\mathbf{list}(\tau)} v$.

Otherwise there exist morphisms $y_1 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$, $y_2 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\mathbf{list}\,(\tau)]\!]$ and values $v_1 : \tau$, $v_2 : \mathbf{list}\,(\tau)$ such that $y = \mathsf{cons} \circ \langle y_1, y_2\rangle$, $v = v_1{::}v_2$, $y_1 \precsim^{\mathrm{V}*}_{\tau} v_1$, and $y_2 \precsim^{\mathrm{V}*}_{\mathbf{list}(\tau)} v_2$. Furthermore, as v_1 and v_2 are values, there exist morphisms $y'_1 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$ and $y'_2 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\mathbf{list}(\tau)]\!]$ such that $\mathcal{V}_{\mathrm{E}}[\![v_1 : \tau]\!] = \mathsf{up} \circ y'_1$ and $\mathcal{V}_{\mathrm{E}}[\![v_2 : \mathbf{list}(\tau)]\!] = \mathsf{up} \circ y'_2$.

By the definition of $\precsim^{\mathrm{V}*}_{\mathbf{list}(\tau)}$, we know that

$$\mathsf{up} \circ \mathsf{List}\,(!) \circ y_2 \leq L(\mathsf{List}\,(!)) \circ \mathcal{V}_{\mathrm{E}}[\![v_2 : \mathbf{list}(\tau)]\!] = \mathsf{up} \circ \mathsf{List}\,(!) \circ y'_2$$

so $\mathsf{List}\,(!) \circ y_2 \leq \mathsf{List}\,(!) \circ y'_2$. Therefore

$$\begin{array}{rcl} \mathsf{up} \circ \mathsf{List}\,(!) \circ y & = & \mathsf{up} \circ \mathsf{List}\,(!) \circ \mathsf{cons} \circ \langle y_1, y_2 \rangle \\ & = & \mathsf{up} \circ \mathsf{cons} \circ \langle !, \mathsf{List}\,(!) \circ y_2 \rangle \\ & \leq & \mathsf{up} \circ \mathsf{cons} \circ \langle !, \mathsf{List}\,(!) \circ y'_2 \rangle \\ & = & \mathsf{up} \circ \mathsf{List}\,(!) \circ \mathsf{cons} \circ \langle y'_1, y'_2 \rangle \\ & = & L(\mathsf{List}\,(!)) \circ \mathcal{V}_{\mathrm{E}}[\![v_1{::}v_2 : \mathbf{list}(\tau)]\!] \end{array}$$

Furthermore,

$$\begin{array}{rcl} p_{L_1} \circ y & = & \mathsf{hd} \circ \mathsf{cons} \circ \langle y_1, y_2 \rangle \\ & = & \mathsf{up} \circ y_1 \\ & \precsim^{\mathrm{V}}_{\tau} & v_1 \end{array}$$

As $\mathtt{head}(v) \Rightarrow_{\mathrm{v}} v_1$, this means that $p_{L_1} \circ y \precsim^{\mathrm{V}}_{\tau} [v/x]e_{L_1} \equiv \mathtt{head}(v)$.

Next let $k > 1$. Then

$$\begin{array}{rcl} p_{L_k} \circ y & = & {p_{L_k - 1}}^{\perp} \circ \mathsf{tl} \circ \mathsf{cons} \circ \langle y_1, y_2 \rangle \\ & = & {p_{L_k - 1}}^{\perp} \circ \mathsf{up} \circ y_2 \\ & = & p_{L_k - 1} \circ y_2 \\ & \precsim^{\mathrm{V}}_{\tau} & [v_2/x]e_{L_{k-1}} \equiv \mathtt{head}(\mathtt{tail}^{k-2}(v_2)) \end{array}$$

Suppose there exists a value v' such that $\mathtt{head}(\mathtt{tail}^{k-2}(v_2)) \Rightarrow_{\mathrm{v}} v'$. Because the only derivation possible for $\mathtt{head}(\mathtt{tail}^{k-2}(v_2)) \Rightarrow_{\mathrm{v}} v'$ is

$$\frac{\mathtt{head} \Rightarrow_{\mathrm{v}} \mathtt{head} \quad \mathtt{tail}^{k-2}(v_2) \Rightarrow_{\mathrm{v}} v'{::}v'' \quad \mathrm{vapply}(\mathtt{head}, v'{::}v'') \Rightarrow v'}{\mathtt{head}(\mathtt{tail}^{k-2}(v_2)) \Rightarrow_{\mathrm{v}} v'}$$

for some value v'', by Lemma A.2.3 we know that the following derivation exists:

$$\frac{\mathtt{head} \Rightarrow_{\mathrm{v}} \mathtt{head} \quad \mathtt{tail}^{k-1}(v_1{::}v_2) \Rightarrow_{\mathrm{v}} v'{::}v'' \quad \mathrm{vapply}(\mathtt{head}, v'{::}v'') \Rightarrow v'}{\mathtt{head}(\mathtt{tail}^{k-1}(v_1{::}v_2)) \Rightarrow_{\mathrm{v}} v'}$$

Thus by Lemma A.1.1, $p_{L_k} \circ y \precsim^{\mathrm{V}}_{\tau} [v/x]e_{L_k}$. Therefore $y \precsim^{\mathrm{V}*}_{\mathbf{list}(\tau)} v$.

□

What remains to be done is to prove that the list constants are adequate:

`nil`: Follows directly from Lemma A.2.4.

`cons`: Let $y_1 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\tau]\!]$ and $y_2 : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}_{\mathrm{E}}[\![\mathbf{list}(\tau)]\!]$. Furthermore suppose v_1 and v_2 are closed values of type τ and $\mathbf{list}(\tau)$ respectively such that $y_1 \precsim^{\mathrm{V}*}_{\tau} v_1$ and $y_2 \precsim^{\mathrm{V}*}_{\mathbf{list}(\tau)} v_2$. Then `cons` is adequate if

$$\mathcal{C}^{\mathrm{V}}_{\mathrm{E}}[\![\mathtt{cons}]\!] \circ \langle\rangle(y_1, y_2) \precsim^{\mathrm{V}}_{\mathbf{list}(\tau)} v_1{::}v_2$$

However,

$$\mathcal{V}_{\mathrm{E}}[\![\mathtt{cons}]\!] \circ \langle\rangle(y_1, y_2) = \mathsf{up} \circ \mathsf{cons} \circ \langle y_1, y_2 \rangle$$

Therefore it follows directly from Lemma A.2.4 that `cons` is adequate.

head: Let y be a morphism from $\mathbf{1}$ to $\mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{list}(\tau)]\!]$ and v be a closed value of type $\mathbf{list}(\tau)$ such that $y \precsim^{\mathrm{V}^*}_{\mathbf{list}(\tau)} v$. If $y = \mathsf{nil}$, then

$$\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{head}]\!] \circ \langle\rangle(y) = \mathsf{hd} \circ \mathsf{nil} = \bot \precsim^{\mathrm{V}}_{\tau} \mathtt{head}(v)$$

Otherwise, by Lemma A.2.4, there exist morphisms $y_1 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$, $y_2 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{list}(\tau)]\!]$ and values $v_1 : \tau$, $v_2 : \mathbf{list}(\tau)$, such that $y = \mathsf{cons} \circ \langle y_1, y_2\rangle$, $v = v_1{::}v_2$, $y_1 \precsim^{\mathrm{V}^*}_{\tau} v_1$ and $y_2 \precsim^{\mathrm{V}^*}_{\mathbf{list}(\tau)} v_2$. Therefore, as

$$\begin{aligned} \mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{head}]\!] \circ \langle\rangle(y) &= \mathsf{hd} \circ \mathsf{cons} \circ \langle y_1, y_2\rangle \\ &= \mathsf{up} \circ y_1 \end{aligned}$$

and $\mathtt{head}(v) \Rightarrow_{\mathrm{v}} v_1$, $\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{head}]\!] \circ \langle\rangle(y) \precsim^{\mathrm{V}}_{\tau} \mathtt{head}(v_1)$. Thus `head` is adequate.

tail: Because of Lemma A.2.4, the proof that `tail` is similar to the proof that `head` is adequate.

nil?: Let $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{list}(\tau)]\!]$ and v be a closed value of type $\mathbf{list}(\tau)$ such that $y \precsim^{\mathrm{V}^*}_{\mathbf{list}(\tau)} v$. If $y = \mathsf{nil}$ then v must be `nil`. Therefore

$$\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{nil?}]\!] \circ \langle\rangle(y) = \mathsf{up} \circ \mathsf{null?} \circ \mathsf{nil} = \mathsf{up} \circ \mathsf{tt}$$

As $\mathsf{tt} \precsim^{\mathrm{V}^*}_{\mathbf{bool}}$ `true` and $\mathtt{nil?}(\mathtt{nil}) \Rightarrow_{\mathrm{n}}$ `true`, it must hold that $\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{nil?}]\!] \circ \langle\rangle(y) \precsim^{\mathrm{V}}_{\mathbf{bool}} \mathtt{nil?}(\mathtt{nil})$.

If $y \neq \mathsf{nil}$ then there exist morphisms $y_1 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\tau]\!]$ and $y_2 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{V}}[\![\mathbf{list}(\tau)]\!]$ such that $y = \mathsf{cons} \circ \langle y_1, y_2\rangle$. Therefore there must also exist values v_1 and v_2 such that $v = v_1{::}v_2$. Thus $\mathtt{nil?}(v) \Rightarrow_{\mathrm{n}} \mathtt{false}$, and

$$\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{nil?}]\!] \circ \langle\rangle(y) = \mathsf{up} \circ \mathsf{null?} \circ \mathsf{cons} \circ \langle y_1, y_2\rangle = \mathsf{up} \circ \mathsf{ff}$$

As $\mathsf{ff} \precsim^{\mathrm{V}^*}_{\mathbf{bool}}$ `false`, $\mathcal{C}_{\mathrm{E}}^{\mathrm{V}}[\![\mathtt{nil?}]\!] \circ \langle\rangle(y) \precsim^{\mathrm{V}}_{\mathbf{bool}} \mathtt{nil?}(v)$. Therefore `nil?` is adequate.

Appendix B

Adequacy of the call-by-name extensional semantics

B.1 Adequacy of the call-by-name semantics

Definition B.1.1 For each type τ, let $\precsim_\tau^{\mathrm{N}}$ be a relation between morphisms from $\mathbf{1}$ to $L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau]\!]$ and closed expressions of type τ, and $\precsim_\tau^{\mathrm{N}^*}$ be a relation between morphisms from $\mathbf{1}$ to $\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau]\!]$ and closed values of type τ, defined as follows: given $z : \mathbf{1} \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau]\!]$ and $\vdash e : \tau$, $z \precsim_\tau^{\mathrm{N}} e$ if

1. $z = \perp_{\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau]\!]}$, or

2. There exists a closed value v of type τ and a morphism $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau]\!]$ such that $z = \mathsf{up} \circ y$, $e \Rightarrow_{\mathrm{n}} v$, and $y \precsim_\tau^{\mathrm{N}^*} v$, where

   - $y \precsim_g^{\mathrm{N}^*} v$ if $\mathsf{up} \circ y \leq \mathcal{N}_{\mathrm{E}}[\![v : \tau]\!]$
   - $y \precsim_{\delta(\tau_1,\ldots,\tau_n)}^{\mathrm{N}^*} v$ if, for each deconstructor $p \in P_i$, $\mathsf{down} \circ p \circ y \precsim_{\tau_i}^{\mathrm{N}} [v/x]e_p$ and if
   $$F_\delta(!,\ldots,!) \circ y \leq \mathcal{K}_{F_\delta}[\![v : \delta(\tau_1,\ldots,\tau_n)]\!]$$
   - $y \precsim_{\tau' \to \tau}^{\mathrm{N}^*} v$ if for all $z' : \mathbf{1} \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau']\!]$ and closed expressions e' such that $z' \precsim_{\tau'}^{\mathrm{N}} e'$,
   $$\mathsf{app} \circ \langle y, z' \rangle \precsim_\tau^{\mathrm{N}} v(e')$$

For any $c \in \mathbf{Const}_{\tau_1 \to \ldots \to \tau_n \to \tau}$ of arity n, a morphism $f : \times(L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau_1]\!], \ldots, L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau_n]\!]) \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau]\!]$ is *adequate for* c if for all morphisms $z_i : \mathbf{1} \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau_i]\!]$ $(1 \leq i \leq n)$ and closed expressions e_i of type τ_i such that $z_i \precsim_{\tau_i}^{\mathrm{N}} e_i$,
$$f \circ \langle\rangle(z_1, \ldots, z_n) \precsim_\tau^{\mathrm{N}} ce_1 \ldots e_n$$
A constant c is *adequate* if $\mathcal{C}_{\mathrm{E}}^{\mathrm{N}}[\![c]\!]$ is adequate for c. It is clear that both `true` and `false` are adequate. As with the call-by-value adequacy proof, if $z \precsim_\tau^{\mathrm{N}} e$, and if whenever there exists a v such that $e \Rightarrow_{\mathrm{n}} v$, $e' \Rightarrow_{\mathrm{n}} v$, then $z \precsim_\tau^{\mathrm{N}} e'$.

Lemma B.1.1 *Suppose that* $\Gamma \vdash e : \tau$, *where* $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$. *Furthermore suppose that for* $1 \leq i \leq n$, *there exist* $z_i : \mathbf{1} \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau_i]\!]$ *and closed expressions* e_i *of type* τ_i *such that* $z_i \precsim_{\tau_i}^{\mathrm{N}} e_i$. *Then*
$$\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e : \tau]\!] \circ \langle\rangle(z_1, \ldots, z_n) \precsim_\tau^{\mathrm{N}} [e_1/x_1, \ldots, e_n/x_n]e$$

Proof. By induction on the structure of e. Let $[s]$ represent the substitution $[e_1/x_1, \ldots, e_n/x_n]$ and let $p_0 = \langle\rangle(f_1, \ldots, f_n)$. All of the cases except those of abstraction and application are essentially the same as in the call-by-value proof so only those cases are included here.

Case $e = \mathtt{lam}\ x.e'$, $\tau = \tau'' \to \tau'$:

Let $f : L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau'']\!] \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau']\!]$ be the morphism

$$f = \mathcal{N}_{\mathrm{E}}[\![\Gamma, x : \tau'' \vdash e' : \tau']\!] \circ \langle p_0 \circ !, \mathsf{id}\rangle$$

Also let $y = \mathsf{curry}(\mathcal{N}_{\mathrm{E}}[\![\Gamma, x : \tau'' \vdash e' : \tau']\!]) \circ p_0$. Then $\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e : \tau : \circ]\!]p_0 = \mathsf{up} \circ y$, so it is sufficent to show that $y \precsim^{\mathrm{N}^*}_{\tau} \mathtt{lam}\ x.e'$.

Let $z'' : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau'']\!]$ and let e'' be a closed expression of type τ'' such that $z'' \precsim^{\mathrm{N}}_{\tau''} e''$. Then

$$\begin{array}{rcl}
\mathsf{app} \circ \langle y, z''\rangle & = & \mathsf{app} \circ \langle \mathsf{curry}(\mathcal{N}_{\mathrm{E}}[\![\Gamma, x : \tau'' \vdash e' : \tau']\!]) \circ p_0, z''\rangle \\
& = & \mathcal{N}_{\mathrm{E}}[\![\Gamma, x : \tau'' \vdash e' : \tau']\!] \circ \langle p_0, z''\rangle \\
& = & \mathcal{N}_{\mathrm{E}}[\![\Gamma, x : \tau'' \vdash e' : \tau']\!] \circ \langle p_0 \circ ! \circ z'', z''\rangle \\
& = & \mathcal{N}_{\mathrm{E}}[\![\Gamma, x : \tau'' \vdash e' : \tau']\!] \circ \langle p_0 \circ !, \mathsf{id}\rangle \circ z'' \\
& = & f \circ z''
\end{array}$$

By the induction hypothesis

$$\begin{array}{rcl}
f \circ z'' & \precsim^{\mathrm{N}}_{\tau'} & [s][e''/x]e' \\
& \equiv & [e''/x]([s]e')
\end{array}$$

Suppose there exists a value v such that $[e''/x]([s]e') \Rightarrow_{\mathrm{n}} v$. Because $\mathtt{lam}\ x.[s]e' \Rightarrow_{\mathrm{n}} \mathtt{lam}\ x.[s]e'$, by the operational rules for application, $(\mathtt{lam}\ x.[s]e')(e'') \Rightarrow_{\mathrm{n}} v$ as well. Therefore

$$\mathsf{app} \circ \langle y, z''\rangle = f \circ z'' \precsim^{\mathrm{N}}_{\tau'} (\mathtt{lam}\ x.[s]e')(e'') \equiv ([s]\mathtt{lam}\ x.e')(e'')$$

Thus, by the definition of $\precsim^{\mathrm{N}^*}_{\tau'' \to \tau'}$, $\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash \mathtt{lam}\ x.e' : \tau'' \to \tau']\!] \circ p_0 \precsim^{\mathrm{N}}_{\tau'' \to \tau'} [s]\mathtt{lam}\ x.e'$.

Case $e = e_1(e_2)$:

Because $\Gamma \vdash e_1(e_2) : \tau$, there exists a type τ' such that $\Gamma \vdash e_1 : \tau' \to \tau$ and $\Gamma \vdash e_2 : \tau'$. By the induction hypothesis $\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!] \circ p_0 \precsim^{\mathrm{N}}_{\tau' \to \tau} [s]e_1$. If $\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!] \circ p_0 = \bot$, then

$$\begin{array}{rcl}
\multicolumn{3}{l}{\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1(e_2) : \tau]\!] \circ p_0} \\
& = & \mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!], \mathsf{up} \circ \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!]\rangle \circ p_0 \\
& = & \mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle \bot, \mathsf{up} \circ \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!] \circ p_0\rangle \\
& = & \bot \\
& \precsim^{\mathrm{N}}_{\tau} & [s]e_1(e_2)
\end{array}$$

Therefore suppose that $\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!] \circ p_0 \neq \bot$. Then there exists a value v_1 and a morphism $y : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau' \to \tau]\!]$ such that $\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!] \circ p_0 = \mathsf{up} \circ y$, $[s]e_1 \Rightarrow_{\mathrm{n}} v_1$, and $y \precsim^{\mathrm{N}^*}_{\tau' \to \tau} v_1$. Also $\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau']\!] \circ p_0 \precsim^{\mathrm{N}}_{\tau'} [s]e_2$, so by the definition of $\precsim^{\mathrm{N}^*}$ for function types,

$$\begin{array}{rcl}
\multicolumn{3}{l}{\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1(e_2) : \tau]\!] \circ p_0} \\
& = & \mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1 : \tau' \to \tau]\!] \circ p_0, \mathsf{up} \circ \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!] \circ p_0\rangle \\
& = & \mathsf{app} \circ \langle y, \mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_2 : \tau]\!] \circ p_0\rangle \\
& \precsim^{\mathrm{N}}_{\tau} & v_1([s]e_2)
\end{array}$$

Suppose there exists a v such that $v_1([s]e_2) \Rightarrow_{\mathrm{n}} v$.

There are two possible derivations of $v_1([s]e_2) \Rightarrow_{\mathrm{n}} v$. The first, where v_1 is of the form $\mathtt{lam}\ x.e'$, is the derivation

$$\frac{v_1 \Rightarrow_{\mathrm{n}} \mathtt{lam}\ x.e' \quad [[s]e_2/x]e' \Rightarrow_{\mathrm{n}} v}{v_1([s]e_2) \Rightarrow_{\mathrm{n}} v}$$

Because $[s]e_1 \Rightarrow_{\mathrm{n}} v_1$ the same application rule can be used to derive that $([s]e_1)([s]e_2) \Rightarrow_{\mathrm{n}} v$, i.e. that $[s]e_1(e_2) \Rightarrow_{\mathrm{n}} v$.

Otherwise v_1 must equal $ce'_1 \ldots e'_i$ for some constant c and expressions $e'_1, \ldots e'_i$ and the derivation is of the form

$$\frac{v_1 \Rightarrow_{\mathrm{n}} ce'_1 \ldots e'_i \quad \mathrm{napply}(c, e'_1, \ldots, e'_i, [s]e_2) \Rightarrow v}{v_1([s]e_2) \Rightarrow_{\mathrm{n}} v}$$

Again as $[s]e_1 \Rightarrow_{\mathrm{n}} v_1$ the same rule can be used to derive that $[s]e_1(e_2) \Rightarrow_{\mathrm{n}} v$. Therefore either way $\mathcal{N}_{\mathrm{E}}[\![\Gamma \vdash e_1(e_2) : \tau]\!] \circ p_0 \precsim^{\mathrm{N}}_{\tau} [s]e_1(e_2)$.

□

Corollary B.1.2 *Suppose that $\vdash e : \tau$. Then $\mathcal{N}_{\mathrm{E}}[\![e : \tau]\!] \neq \bot$ if and only if there exists a value v such that $e \Rightarrow_{\mathrm{n}} v$.*

Proof. If $\mathcal{N}_{\mathrm{E}}[\![e : \tau]\!] \neq \bot$ then as $\mathcal{N}_{\mathrm{E}}[\![e]\!] \precsim^{\mathrm{N}}_{\tau} e$, there exists a value v such that $e \Rightarrow_{\mathrm{n}} v$. If $e \Rightarrow_{\mathrm{n}} v$ then by soundness $\mathcal{N}_{\mathrm{E}}[\![e : \tau]\!] = \mathcal{N}_{\mathrm{E}}[\![v : \tau]\!]$ and because $\mathcal{N}_{\mathrm{E}}[\![v : \tau]\!]$ is a value, it factors through up so does not equal $\bot$. □

B.2 Adequacy of the FL constants

B.2.1 Ground type constants

As with the call-by-value semantics it is possible to show that the ground type constants are immediately adequate. It is also possible to show that constants representing functions on ground types (and returning a ground type) are also adequate.

Theorem B.2.1 *Suppose that $c \in \mathcal{T}^{\mathrm{N}}[\![\tau_1 \to \ldots \to \tau_n \to \tau]\!]$ has arity n. Furthermore, suppose that all types $\tau_1, \ldots, \tau_n$, and τ are ground types. Then $\mathcal{C}^{\mathrm{N}}_{\bot}[\![c]\!]$ is adequate for c.*

Proof. For $1 \leq i \leq n$, let $z_i : \mathbf{1} \to L\mathcal{T}^{\mathrm{v}}_{\bot}[\![\tau_i]\!]$ and let e_i be a closed expression of type τ_i such that $z_i \precsim^{\mathrm{N}}_{\tau_i} e_i$. Because each τ_i is a ground type, for each i, $z_i \leq \mathcal{N}_{\bot}[\![e_i : \tau_i]\!]$. Therefore

$$\begin{array}{rcl} \mathcal{C}^{\mathrm{N}}_{\bot}[\![c]\!] \circ \langle\rangle(z_1, \ldots, z_n) & = & \mathsf{raise}^{(n-n)}_{\bot}(\mathcal{C}^{\mathrm{N}}_{\bot}[\![c]\!]) \circ \langle\rangle(z_1, \ldots, z_n) \\ & \leq & \mathsf{raise}^{(n-n)}_{\bot}(\mathcal{C}^{\mathrm{N}}_{\bot}[\![c]\!]) \circ \langle\rangle(\mathcal{N}_{\bot}[\![e_1 : \tau_1]\!], \ldots, \mathcal{N}_{\bot}[\![e_n : \tau_n]\!]) \\ & = & \mathcal{N}_{\bot}[\![ce_1 \ldots e_n : \tau]\!] \\ & \precsim^{\mathrm{N}}_{\tau} & ce_1 \ldots e_n \end{array}$$

Thus because τ is a ground type, by the definition of $\precsim^{\mathrm{N}}$ and $\precsim^{\mathrm{N}^*}$ for ground types,

$$\mathcal{C}^{\mathrm{N}}_{\bot}[\![c]\!] \circ \langle\rangle(z_1, \ldots, z_n) \precsim^{\mathrm{N}}_{\tau} ce_1 \ldots e_n$$

Therefore $\mathcal{C}^{\mathrm{N}}_{\perp}[\![c]\!]$ is adequate for c. □

All the integer (and boolean) constants satisfy the requirements for the theorem and thus are all adequate.

As we did for the call-by-value case, we will first show that the definition of adequacy for the product, sum, and list constants is equivalent to a more intuitive definition (which will not need the structure function $\mathcal{K}[\![-]\!]$). We then use the more intuitive definition to show that the FL constants are adequate.

B.2.2 Products

For products, we want the structure function to satisfy the following equation for any closed value v of type $\tau_1 \times \tau_2$:

$$(! \times !) \circ y = \mathcal{K}_{\times}[\![v : \tau_1 \times \tau_2]\!]$$

where $\mathsf{up} \circ y = \mathcal{N}_{\mathrm{E}}[\![v : \tau_1 \times \tau_2]\!]$. However, $(! \times !) \circ y = \langle !, ! \rangle$, therefore let $\mathcal{K}_{\times}[\![v : \tau_1 \times \tau_2]\!] = \langle !, ! \rangle$.

Lemma B.2.2 *For any $y : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 \times \tau_2]\!]$, $y \precsim^{\mathrm{N}^*}_{\tau_1 \times \tau_2} v$ if and only if there exist morphisms $z_1 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$, $z_2 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]$ and values $v_1 : \tau_1$, $v_2 : \tau_2$ such that $y = \langle z_1, z_2 \rangle$, $v = \langle v_1, v_2 \rangle$, $z_1 \precsim^{\mathrm{N}}_{\tau_1} v_1$, and $z_2 \precsim^{\mathrm{N}}_{\tau_2} v_2$.*

Proof.

Case $\Rightarrow$: Suppose that $y \precsim^{\mathrm{N}^*}_{\tau_1 \times \tau_2} v$. Then because y is a morphism from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 \times \tau_2]\!] = L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!] \times L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]$, there must exist morphisms $z_1 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$ and $z_2 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]$ such that $y = \langle z_1, z_2 \rangle$ (namely $z_1 = \pi_1 \circ y$ and $z_2 = \pi_2 \circ y$). Because $\mathsf{down} \circ p_{\times_1} = \mathsf{down} \circ \mathsf{up} \circ \pi_1 = \pi_1$ and $\mathsf{down} \circ p_{\times_2} = \pi_2$, we immediately know that

$$z_1 = \pi_1 \circ y \precsim^{\mathrm{N}}_{\tau_1} \mathtt{fst}(v)$$

and

$$z_2 = \pi_2 \circ y \precsim^{\mathrm{N}}_{\tau_2} \mathtt{snd}(v)$$

Lastly, by the definition of values, there must exist values v_1 and v_2 such that $v = \langle v_1, v_2 \rangle$. Therefore $\mathtt{fst}(v) \Rightarrow_{\mathrm{n}} v_1$ and $\mathtt{snd}(v) \Rightarrow_{\mathrm{n}} v_2$, so $z_1 \precsim^{\mathrm{N}}_{\tau_1} v_1$ and $z_2 \precsim^{\mathrm{N}}_{\tau_2}$.

Case $\Leftarrow$: Suppose that there exist morphisms $z_1 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$, $z_2 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]$ and values $v_1 : \tau_1$, $v_2 : \tau_2$ such that $y = \langle z_1, z_2 \rangle$, $v = \langle v_1, v_2 \rangle$, $z_1 \precsim^{\mathrm{N}}_{\tau_1} v_1$, and $z_2 \precsim^{\mathrm{N}}_{\tau_2} v_2$. We need to show that $y \precsim^{\mathrm{N}^*}_{\tau_1 \times \tau_2} v$.

Because $\langle !, ! \rangle$ is the only morphism from $\mathbf{1}$ to $\mathbf{1} \times \mathbf{1}$, for any morphism $y : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 \times \tau_2]\!]$,

$$(! \times !) \circ y = \mathcal{K}_{\times}[\![v : \tau_1 \times \tau_2]\!]$$

Furthermore, as noted above,

$$\mathsf{down} \circ p_{\times_1} \circ y = \pi_1 \circ y = z_1 \precsim^{\mathrm{N}}_{\tau_1} v_1$$

and

$$\mathsf{down} \circ p_{\times_2} \circ y = \pi_2 \circ y = z_2 \precsim^{\mathrm{N}}_{\tau_2} v_2$$

As $\mathtt{fst}(v) \Rightarrow_{\mathrm{n}} v_1$ and $\mathtt{snd}(v) \Rightarrow_{\mathrm{n}} v_2$, we know that

$$\mathsf{down} \circ p_{\times_1} \circ y \precsim^{\mathrm{N}}_{\tau_1} \mathtt{fst}(v)$$

and

$$\mathsf{down} \circ p_{\times_2} \circ y \precsim^{\mathrm{N}}_{\tau_2} \mathtt{fst}(v)$$

Therefore $y \precsim^{\mathrm{N}^*}_{\tau_1 \times \tau_2} v$.

□

We can now check that the constants are adequate. For $\mathtt{pair}$, let z_1 be a morphism from $\mathbf{1}$ to $L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$, let z_2 be a morphism from $\mathbf{1}$ to $L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]$, let e_1 be a closed expression of type τ_1 and let e_2 be a closed expression of type τ_2 such that $z_1 \precsim^{\mathrm{N}}_{\tau_1} e_1$ and $z_2 \precsim^{\mathrm{N}}_{\tau_2} e_2$. Now suppose that either z_1 or z_2 is $\perp$. Then $\mathsf{nsmash}(2) \circ \langle\rangle(z_1, z_2)$ is $\perp$, so

$$\begin{aligned}
\mathcal{C}^{\mathrm{N}}_{\perp}[\![\mathtt{pair}]\!] \circ \langle\rangle(z_1, z_2) &= (\mathsf{up} \circ (\mathsf{up} \times \mathsf{up}) \circ (\pi_2 \times \mathsf{id}))^{\perp} \circ \mathsf{nsmash}(2) \circ \langle\rangle(z_1, z_2) \\
&= (\mathsf{up} \circ (\mathsf{up} \times \mathsf{up}) \circ (\pi_2 \times \mathsf{id}))^{\perp} \circ \perp \\
&= \perp
\end{aligned}$$

Therefore $\mathcal{C}^{\mathrm{N}}_{\perp}[\![\mathtt{pair}]\!] \circ \langle\rangle(z_1, z_2) \precsim^{\mathrm{N}}_{\tau_1 \times \tau_2} \mathtt{pair}\ e_1 e_2$.

Otherwise there exists $y_1' : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$ and $y_2' : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]$ such that $z_1 = \mathsf{up} \circ y_1'$ and $z_2 = \mathsf{up} \circ y_2'$. Because $z_1 \precsim^{\mathrm{N}}_{\tau_1} e_1$ and $z_2 \precsim^{\mathrm{N}}_{\tau_2} e_2$ there exists values v_1, v_2 such that $e_1 \Rightarrow_{\mathrm{n}} v_1$, $e_2 \Rightarrow_{\mathrm{n}} v_2$, $y_1' \precsim^{\mathrm{N}^*}_{\tau_1} v_1$, and $y_2' \precsim^{\mathrm{N}}_{\tau_2} v_2$. Therefore by the above properties of products

$$\begin{aligned}
\mathcal{C}^{\mathrm{N}}_{\perp}[\![\mathtt{pair}]\!] \circ \langle\rangle(z_1, z_2) &= (\mathsf{up} \circ (\mathsf{up} \times \mathsf{up}) \circ (\pi_2 \times \mathsf{id}))^{\perp} \circ \mathsf{nsmash}(2) \circ \langle\rangle(\mathsf{up} \circ y_1', \mathsf{up} \circ y_2') \\
&= (\mathsf{up} \circ (\mathsf{up} \times \mathsf{up}) \circ (\pi_2 \times \mathsf{id}))^{\perp} \circ \mathsf{up} \circ \langle\rangle(y_1', y_2') \\
&= \mathsf{up} \circ (\mathsf{up} \times \mathsf{up}) \circ (\pi_2 \times \mathsf{id}) \circ \langle\rangle(y_1', y_2') \\
&= \mathsf{up} \circ \langle \mathsf{up} \circ y_1', \mathsf{up} \circ y_2' \rangle \\
&= \mathsf{up} \circ \langle z_1, z_2 \rangle \\
&\precsim^{\mathrm{N}}_{\tau_1 \times \tau_2} <v_1, v_2>
\end{aligned}$$

With the following derivation

$$\frac{e_1 \Rightarrow_{\mathrm{n}} v_1 \quad e_2 \Rightarrow_{\mathrm{n}} v_2}{\mathrm{napply}(\mathtt{pair}, e_1, e_2) \Rightarrow <v_1, v_2>}$$

we know that $<e_1, e_2> \Rightarrow_{\mathrm{n}} <v_1, v_2>$. Therefore

$$\mathcal{C}^{\mathrm{N}}_{\perp}[\![\mathtt{pair}]\!] \circ \langle\rangle(z_1, z_2) \precsim^{\mathrm{N}}_{\tau_1 \times \tau_2} <e_1, e_2>$$

so $\mathtt{pair}$ is adequate.

For $\mathtt{fst}$, let $z : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 \times \tau_2]\!]$ and e be a closed expression of type $\tau_1 \times \tau_2$ such that $z \precsim^{\mathrm{N}}_{\tau_1 \times \tau_2} e$. If $z = \perp$ then

$$\mathcal{C}^{\mathrm{N}}_{\perp}[\![\mathtt{fst}]\!] \circ \langle\rangle(z) = {\pi_1}^{\perp} \circ \pi_2 \circ \langle\rangle(\perp) = \perp$$

so $\mathcal{C}^{\mathrm{N}}_{\perp}[\![\mathtt{fst}]\!] \circ \langle\rangle(z) \precsim^{\mathrm{N}}_{\tau_1} \mathtt{fst}(e)$. Otherwise there exists a $y : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 \times \tau_2]\!]$ such that $z = \mathsf{up} \circ y$ and there exists a closed value v of type $\tau_1 \times \tau_2$ such that $e \Rightarrow_{\mathrm{n}} v$ and $y \precsim^{\mathrm{N}^*}_{\tau_1 \times \tau_2} v$. Thus by Lemma B.2.2 there exist morphisms $z_1 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$, $z_2 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}[\![\tau_2]\!]$, and closed values v_1 and v_2 of types τ_1 and τ_2, respectively, such that $v = <v_1, v_2>$, $y = \langle z_1, z_2 \rangle$, and $z_1 \precsim^{\mathrm{N}}_{\tau_1} v_1$. Therefore

$$\mathcal{C}^{\mathrm{N}}_{\perp}[\![\mathtt{fst}]\!] \circ \langle\rangle(z) = {\pi_1}^{\perp} \circ \pi_2 \circ \langle\rangle(\mathsf{up} \circ \langle z_1, z_2 \rangle) = z_1 \precsim^{\mathrm{N}}_{\tau_1} v_1$$

As $\mathtt{fst}(v) \Rightarrow_{\mathrm{n}} v_1$, we know that $\mathcal{C}^{\mathrm{N}}_{\perp}[\![\mathtt{fst}]\!] \circ \langle\rangle(z) \precsim^{\mathrm{N}}_{\tau_1} \mathtt{fst}(v)$ as well, therefore $\mathtt{fst}$ is adequate.

The proof that $\mathtt{snd}$ is adequate is similar.

B.2.3 Sums

The deconstructors from the call-by-value adequacy proof were $p_{+_1} = [\mathsf{up}, \bot \circ !]$ and $p_{+_2} = [\bot \circ !, \mathsf{up}]$ with

$$e_{+_1} = \texttt{case}\ x\ \texttt{of}\ \texttt{left} : \texttt{lam}\ y.y\ \texttt{right} : \texttt{rec}\ z.z$$

and

$$e_{+_2} = \texttt{case}\ x\ \texttt{of}\ \texttt{left} : \texttt{rec}\ z.z\ \texttt{right} : \texttt{lam}\ y.y$$

as the related expressions. When combined with down, the deconstructors become $[\mathsf{id}, \bot \circ !]$ and $[\bot \circ !, \mathsf{id}]$.

For the structure function, let v be a value of type $\tau_1 + \tau_2$. Then the only possible definitions for $\mathcal{K}_+[\![-]\!]$ such that $L(! + !) \circ \mathcal{N}_{\mathrm{E}}[\![v : \tau_1 + \tau_2]\!] = \mathsf{up} \circ \mathcal{K}_+[\![v : \tau_1 + \tau_2]\!]$ are

$$\mathcal{K}_+[\![\texttt{inl}(v) : \tau_1 + \tau_2]\!] = \mathsf{inl} : \mathbf{1} \to \mathbf{1} + \mathbf{1}$$

and

$$\mathcal{K}_+[\![\texttt{inr}(v) : \tau_1 + \tau_2]\!] = \mathsf{inr} : \mathbf{1} \to \mathbf{1} + \mathbf{1}$$

Suppose that y is a morphism from $\mathbf{1}$ to $\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 + \tau_2]\!]$. Then either $y = \mathsf{inl} \circ z_1$ for some $z_1 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$, or $y = \mathsf{inr} \circ z_2$ for some $z_2 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]$. This leads to the following lemma:

Lemma B.2.3 *Suppose that $y : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 + \tau_2]\!]$ and v is a closed value of type $\tau_1 + \tau_2$. Then $y \precsim^{\mathrm{N}*}_{\tau_1+\tau_2} v$ if and only if one of the following holds:*

- *There exist a morphism $z_1 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$ and a closed expression e_1 of type τ_1 such that $y = \mathsf{inl} \circ z_1$, $v = \texttt{inl}(e_1)$ and $z_1 \precsim^{\mathrm{N}}_{\tau_1} e_1$.*
- *There exist a morphism $z_2 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]$ and a closed expression e_2 of type τ_2 such that $y = \mathsf{inr} \circ z_2$, $v = \texttt{inr}(e_2)$ and $z_2 \precsim^{\mathrm{N}}_{\tau_2} e_2$.*

Proof.

⇒: Suppose that $y \precsim^{\mathrm{N}*}_{\tau_1+\tau_2} v$. If there exists some $z_1 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$ such that $y = \mathsf{inl} \circ z_1$, then

$$! + ! \circ y = \mathsf{inl} = \mathcal{K}_+[\![v : \tau_1 + \tau_2]\!]$$

so there must exist a closed expression e_1 of type τ_1 such that $v = \texttt{inl}(e_1)$. What is left is to show that $z_1 \precsim^{\mathrm{N}}_{\tau_1} e_1$. By the definition of $\precsim^{\mathrm{N}}_{\tau_1+\tau_2}$,

$$\begin{array}{rcl} z_1 & = & [\mathsf{id}, \bot \circ !] \circ \mathsf{inl} \circ z_1 \\ & = & \mathsf{down} \circ p_{+_1} \circ y \\ & \precsim^{\mathrm{N}}_{\tau_1} & \texttt{case}\ v\ \texttt{of}\ \texttt{left} : \texttt{lam}\ y.y\ \texttt{right} : \texttt{rec}\ z.z \end{array}$$

Suppose there exists a value v_1 such that

$$\texttt{case}\ v\ \texttt{of}\ \texttt{left} : \texttt{lam}\ y.y\ \texttt{right} : \texttt{rec}\ z.z \Rightarrow_{\mathrm{n}} v_1$$

Because $v = \texttt{inl}(e_1)$ there is only one possible derivation:

$$\dfrac{v \Rightarrow_{\mathrm{n}} \texttt{inl}(e_1) \qquad \dfrac{\dfrac{\texttt{lam}\ y.y \Rightarrow_{\mathrm{n}} \texttt{lam}\ y.y \quad e_1 \Rightarrow_{\mathrm{n}} v_1}{(\texttt{lam}\ y.y)(e_1) \Rightarrow_{\mathrm{n}} v_1}}{\mathrm{napply}(\texttt{case}, v, \texttt{lam}\ y.y, \texttt{rec}\ z.z) \Rightarrow v_1}}{\texttt{case}\ v\ \texttt{of}\ \texttt{left} : \texttt{lam}\ y.y\ \texttt{right} : \texttt{rec}\ z.z \Rightarrow_{\mathrm{n}} v_1}$$

Therefore $e_1 \Rightarrow_n v_1$, so $z_1 \precsim^{\mathrm{N}}_{\tau_1} e_1$.

If y is not of the form $\mathsf{inl} \circ z_1$ for some z_1, then there must exist a morphism $z_2 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]$ such that $y = \mathsf{inr} \circ z_2$. By a similar proof as above, we can then show that there must exist a closed expression e_2 of type τ_2 such that $v = \mathtt{inr}(e_2)$ and $z_2 \precsim^{\mathrm{N}}_{\tau_2} e_2$. Thus exactly one of the listed properties holds, proving the lemma.

$\Leftarrow$: Suppose that one of the listed properties holds. We need to show that $y \precsim^{\mathrm{N}*}_{\tau_1+\tau_2} v$.

If the first property holds, then there exist a morphism $z_1 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$ and a closed expression e_1 of type τ_1 such that $y = \mathsf{inl} \circ z_1$, $v = \mathtt{inr}(e_1)$, and $z_1 \precsim^{\mathrm{N}}_{\tau_1} e_1$. Thus

$$(! + !) \circ y = \mathsf{inl} = \mathcal{K}_+[\![v : \tau_1 + \tau_2]\!]$$

Furthermore,

$$\mathsf{down} \circ p_{+_1} \circ y = [\mathsf{id}, \perp \circ !] \circ (\text{left} : \circ) z_1 = z_1 \precsim^{\mathrm{N}}_{\tau_1} e_1$$

and

$$\mathsf{down} \circ p_{+_2} \circ y[\perp \circ !, \mathsf{id}] \circ (\text{left} : \circ) z_1 = \perp \precsim^{\mathrm{N}}_{\tau_2} \mathtt{case}\ v\ \mathtt{of}\ \mathtt{left} : \mathtt{rec}\ z.z\ \mathtt{right} : \mathtt{lam}\ y.y$$

so all we need to show is that

$$z_1 \precsim^{\mathrm{N}}_{\tau_1} \mathtt{case}\ v\ \mathtt{of}\ \mathtt{left} : \mathtt{lam}\ y.y\ \mathtt{right} : \mathtt{rec}\ z.z$$

Suppose there exists a value v_1 such that $e_1 \Rightarrow_n v_1$. Then, with the derivation

$$\dfrac{\dfrac{\mathtt{lam}\ y.y \Rightarrow_n \mathtt{lam}\ y.y \quad e_1 \Rightarrow_n v_1}{(\mathtt{lam}\ y.y)(e_1) \Rightarrow_n v_1}}{\text{napply}(\mathtt{case}, v, \mathtt{lam}\ y.y, \mathtt{rec}\ z.z) \Rightarrow v_1}$$

we know that $\mathtt{case}\ v\ \mathtt{of}\ \mathtt{left} : \mathtt{lam}\ y.y\ \mathtt{right} : \mathtt{rec}\ z.z \Rightarrow_n v_1$ as well. Thus

$$z_1 \precsim^{\mathrm{N}}_{\tau_1} \mathtt{case}\ v\ \mathtt{of}\ \mathtt{left} : \mathtt{lam}\ y.y\ \mathtt{right} : \mathtt{rec}\ z.z$$

so $y \precsim^{\mathrm{N}*}_{\tau_1+\tau_2} v$.

Otherwise, the second property must hold, so there exists a morphism $z_2 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_2]\!]$ and a closed expression e_2 of type τ_2 such that $y = \mathsf{inr} \circ z_2$, $v = \mathtt{inr}(e_2)$, and $z_2 \precsim^{\mathrm{N}}_{\tau_2} e_2$. Using an argument similar to the one given for the second property we can show that this implies that $y \precsim^{\mathrm{N}*}_{\tau_1+\tau_2} v$.

□

We can now easily prove that the individual constants are adequate. For $\mathtt{inl}$, let e_1 be a closed expression of type τ_1 and let z be a morphism from $\mathbf{1}$ to $L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1]\!]$ such that $z \precsim^{\mathrm{N}}_{\tau_1} e_1$. Then

$$\mathcal{C}^{\mathrm{N}}[\![\mathtt{inl}]\!] \circ \langle\rangle(z) = \mathsf{up} \circ \mathsf{inl} \circ \pi_2 \circ z = \mathsf{up} \circ \mathsf{inl} \circ z$$

so, by Lemma B.2.3, $\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\mathtt{inl}]\!] \circ \langle\rangle(z) \precsim^{\mathrm{N}}_{\tau_1+\tau_2} \mathtt{inl}(e_1)$. Thus $\mathtt{inl}$ is adequate. Similarly, Lemma B.2.3 ensures that $\mathtt{inr}$ is adequate.

For $\mathtt{case}$, let e, e_1, and e_2 be expressions of type $\tau_1 + \tau_2$, $\tau_1 \to \tau$, and $\tau_2 \to \tau$, respectively, and let z, z_1, and z_2 be morphisms such that $z \precsim^{\mathrm{N}}_{\tau_1+\tau_2} e$, $z_1 \precsim^{\mathrm{N}}_{\tau_1 \to \tau} e_1$, and $z_2 \precsim^{\mathrm{N}}_{\tau_1+\tau_2} e_2$. From the definition of $\mathtt{case}$, we know that

$$\begin{aligned}\mathcal{C}^{\mathrm{N}}[\![\mathsf{case}]\!] \circ \langle\rangle(z, z_1, z_2) &= (\mathsf{case}(\mathsf{nleft}, \mathsf{nright}))^{\perp} \circ \mathsf{smash} \circ (\pi_2 \times \mathsf{up}) \circ \alpha^r \circ \langle\rangle(z, z_1, z_2) \\ &= (\mathsf{case}(\mathsf{nleft}, \mathsf{nright}))^{\perp} \circ \mathsf{smash} \circ \langle z, \mathsf{up} \circ \langle z_1, z_2\rangle\rangle\end{aligned}$$

If $z = \bot$, then

$$\begin{array}{rcl} \mathcal{C}^{\mathrm{N}}[\![\texttt{case}]\!] \circ \langle\rangle(z, z_1, z_2) & = & (\mathsf{case}(\mathsf{nleft}, \mathsf{nright}))^{\bot} \circ \mathsf{smash} \circ \langle \bot, \mathsf{up} \circ \langle z_1, z_2\rangle\rangle \\ & = & (\mathsf{case}(\mathsf{nleft}, \mathsf{nright}))^{\bot} \circ \bot \\ & = & \bot \\ & \precsim^{\mathrm{N}}_{\tau} & \texttt{case}\ e\ \texttt{of}\ \texttt{left} : e_1\ \texttt{right} : e_2 \end{array}$$

If $z \neq \bot$, then there exist a value v and a morphism $y : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 + \tau_2]\!]$ such that $e \Rightarrow_{\mathrm{n}} v$, $z = \mathsf{up} \circ y$, and $y \precsim^{\mathrm{N}^*}_{\tau_1+\tau_2} v$. Then by Lemma B.2.3 there are two possibilities:

- For some z'_1 and e'_1, $y = \mathsf{inl} \circ z'_1$, $v = \texttt{inl}(e'_1)$, and $z'_1 \precsim^{\mathrm{N}}_{\tau_1} e'_1$

  Because $z'_1 \precsim^{\mathrm{N}}_{\tau_1} e'_1$ and $z_1 \precsim^{\mathrm{N}}_{\tau_1\to\tau} e_1$ then either $z_1 = \bot$, in which case

  $$\mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle z_1, \mathsf{up} \circ z'_1\rangle = \bot \precsim^{\mathrm{N}}_{\tau} e_1(e'_1)$$

  or there exist a morphism $y_1 : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau_1 \to \tau]\!]$ such that $z_1 = \mathsf{up} \circ y_1$ and a value v_1 such that $e_1 \Rightarrow_{\mathrm{n}} v_1$ and $y \precsim^{\mathrm{N}^*}_{\tau_1\to\tau} v_1$. In that case

  $$\mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle z_1, \mathsf{up} \circ z'_1\rangle = \mathsf{app} \circ \langle y_1, z'_1\rangle \precsim^{\mathrm{N}}_{\tau} v_1(e'_1)$$

  If there exists a v' such that $v_1(e'_1) \Rightarrow_{\mathrm{n}} v'$ then, as $e_1 \Rightarrow_{\mathrm{n}} v_1$, $e_1(e'_1) \Rightarrow_{\mathrm{n}} v'$ as well. Thus

  $$\mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle z_1, \mathsf{up} \circ z'_1\rangle \precsim^{\mathrm{N}}_{\tau} e_1(e'_1)$$

  Therefore, either way

  $$\begin{array}{rcl} \mathsf{nleft} \circ \langle z'_1, \langle z_1, z_2\rangle\rangle & = & \mathsf{app}^{\bot} \circ \mathsf{smash} \circ \beta \circ (\mathsf{up} \times \pi_1) \circ \langle z'_1, \langle z_1, z_2\rangle\rangle \\ & = & \mathsf{app}^{\bot} \circ \mathsf{smash} \circ \langle z_1, \mathsf{up} \circ z'_1\rangle \\ & \precsim^{\mathrm{N}}_{\tau} & e_1(e'_1) \end{array}$$

  so

  $$\begin{array}{rl} \mathcal{C}^{\mathrm{N}}[\![\texttt{case}]\!] \circ \langle\rangle(z, z_1, z_2) & \\ = & (\mathsf{case}(\mathsf{nleft}, \mathsf{nright}))^{\bot} \circ \mathsf{smash} \circ \langle \mathsf{up} \circ \mathsf{inl} \circ z'_1, \mathsf{up} \circ \langle z_1, z_2\rangle\rangle \\ = & \mathsf{case}(\mathsf{nleft}, \mathsf{nright}) \circ \langle \mathsf{inl} \circ z'_1, \langle z_1, z_2\rangle\rangle \\ = & \mathsf{nleft} \circ \langle z'_1, \langle z_1, z_2\rangle\rangle \\ \precsim^{\mathrm{N}}_{\tau} & e_1(e'_1) \end{array}$$

  By the derivation

  $$\frac{e \Rightarrow_{\mathrm{n}} \texttt{inl}(e'_1) \quad e_1(e'_1) \Rightarrow_{\mathrm{n}} v}{\mathrm{napply}(\texttt{case}, e, e_1, e_2) \Rightarrow v}$$

  it is clear that, for any value v', $e_1(e'_1) \Rightarrow_{\mathrm{n}} v'$ implies that $\texttt{case}\ e\ \texttt{of}\ \texttt{left} : e_1\ \texttt{right} : e_2 \Rightarrow_{\mathrm{n}} v'$. Therefore

  $$\mathcal{C}^{\mathrm{N}}[\![\texttt{case}]\!] \circ \langle\rangle(z, z_1, z_2) \precsim^{\mathrm{N}}_{\tau} \texttt{case}\ e\ \texttt{of}\ \texttt{left} : e_1\ \texttt{right} : e_2$$

- For some z'_2 and e'_2, $y = \mathsf{inr} \circ z'_2$, $v = \texttt{inr}(e'_2)$, and $z'_2 \precsim^{\mathrm{N}}_{\tau_2} e'_2$

  As in the previous case, it can be shown that

  $$\mathsf{nright} \circ \langle\rangle(z'_2, \langle z_1, z_2\rangle) \precsim^{\mathrm{N}}_{\tau} e_2(e'_2)$$

which implies that

$$\mathcal{C}^{\mathrm{N}}[\![\mathtt{case}]\!] \circ \langle\rangle(z, z_1, z_2) \precsim^{\mathrm{N}}_{\tau} e_2(e_2')$$

Also by the derivation

$$\frac{e \Rightarrow_{\mathrm{n}} \mathtt{inr}(e_2') \quad e_2(e_2') \Rightarrow_{\mathrm{n}} v}{\mathrm{napply}(\mathtt{case}, e, e_1, e_2) \Rightarrow v}$$

it holds that for any value v', $e_2(e_2') \Rightarrow_{\mathrm{n}} v'$ implies that $\mathtt{case}\ e\ \mathtt{of}\ \mathtt{left} : e_1\ \mathtt{right} : e_2 \Rightarrow_{\mathrm{n}} v'$ Thus

$$\mathcal{C}^{\mathrm{N}}[\![\mathtt{case}]\!] \circ \langle\rangle(z, z_1, z_2) \precsim^{\mathrm{N}}_{\tau} \mathtt{case}\ e\ \mathtt{of}\ \mathtt{left} : e_1\ \mathtt{right} : e_2$$

Therefore in all cases

$$\mathcal{C}^{\mathrm{N}}[\![\mathtt{case}]\!] \circ \langle\rangle(z, z_1, z_2) \precsim^{\mathrm{N}}_{\tau} \mathtt{case}\ e\ \mathtt{of}\ \mathtt{left} : e_1\ \mathtt{right} : e_2$$

so case is adequate.

B.2.4 Lists

For lazy lists, there is a deconstructor for each potential element of the list, i.e., $p_{\mathrm{L}_1^Z} = \mathsf{lhd}$ and for $k > 1$, $p_{\mathrm{L}_k^Z} = p_{\mathrm{L}_{k-1}^Z}{}^{\perp} \circ \mathsf{ltl}$. The related expressions are $e_{\mathrm{L}_1^Z} = \mathtt{head}(x)$ and, for $k > 1$,

$$e_L = e_{\mathrm{L}_{k-1}^Z}(\mathtt{tail}(x)) \equiv \mathtt{head}(\mathtt{tail}^{(k-1)}(x))$$

Defining the structure function is more complicated, because the tail of a non empty list value, $e_1{::}e_2$ is not a value. If we evaluate e_2, it may itself be non-empty, leading to a possibly infinite chain of evaluations. To prevent this, we first approximate the structure function, ending abruptly after some finite number of iterations. We can then define $\mathcal{K}_{\mathrm{L}^Z}[\![-]\!]$ to be the limit of the approximations.

Therefore for $n \geq 0$, let F_n be the following function from values of list type $\mathbf{list}(\tau)$ to morphisms from $\mathbf{1}$ to $\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\mathbf{list}(\tau)]\!]$:

$$\begin{array}{rcl} F_n[\![\mathtt{nil}]\!] & = & \mathsf{lnil} \\ F_n[\![e_1{::}e_2]\!] & = & \left\{\begin{array}{ll} \mathsf{lcons} \circ \langle !, \mathsf{up} \circ F_{n-1}[\![v_2]\!]\rangle & \exists v_2 . e_2 \Rightarrow_{\mathrm{n}} v_2 \text{ and } n > 0 \\ \mathsf{lcons} \circ \langle !, \perp\rangle & \text{otherwise} \end{array}\right. \end{array}$$

It should be clear that each F_n is well defined and that for any value v that has list type, $F_n[\![v]\!] \leq F_{n+1}[\![v]\!]$. Therefore we can define $\mathcal{K}_{\mathrm{L}^Z}[\![v : \mathbf{list}(\tau)]\!]$ to be $\bigsqcup_{n=0}^{\infty} F_n[\![v]\!]$. $\mathcal{K}_{\mathrm{L}^Z}[\![v : \mathbf{list}(\tau)]\!]$ satisfies the following equation:

$$\begin{array}{rcl} \mathcal{K}_{\mathrm{L}^Z}[\![\mathtt{nil} : \mathbf{list}(\tau)]\!] & = & \mathsf{lnil} \\ \mathcal{K}_{\mathrm{L}^Z}[\![e_1{::}e_2 : \mathbf{list}(\tau)]\!] & = & \left\{\begin{array}{ll} \mathsf{lcons} \circ \langle !, \mathsf{up} \circ \mathcal{K}_{\mathrm{L}^Z}[\![\mathtt{nil} : \mathbf{list}(\tau)]\!]\rangle & \exists v_2 . e_2 \Rightarrow_{\mathrm{n}} v_2 \\ \mathsf{lcons} \circ \langle !, \perp\rangle & \text{otherwise} \end{array}\right. \end{array}$$

We can now prove that the constants are adequate, again by first showing that the generic definition of adequacy is equivalent to a more intuitive definition.

Lemma B.2.4 *Suppose that there exists closed expressions e_1 and e_2 of types τ and $\mathbf{list}(\tau)$, respectively. Then for all $k > 0$, $\mathtt{tail}^{(k)}(e_1{::}e_2) \Rightarrow_{\mathrm{n}} v$ if and only if $\mathtt{tail}^{(k-1)}(e_2) \Rightarrow_{\mathrm{n}} v$.*

Proof. Straightforward by induction on k. □

Lemma B.2.5 *Suppose that e is a closed expression of type* $\mathbf{list}(\tau)$ *and that* $e \Rightarrow_{\mathrm{n}} v$. *Then for all* $k \geq 0$ *and values* v', $\mathtt{tail}^{(k)}(e) \Rightarrow_{\mathrm{n}} v'$ *if and only if* $\mathtt{tail}^{(k)}(v) \Rightarrow_{\mathrm{n}} v'$.

Proof. By straightforward induction on k. □

Theorem B.2.6 *Let* $y : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\mathbf{list}(\tau)]\!]$ *for some type* τ *and suppose that* v *is a closed value of type* $\mathbf{list}(\tau)$. *Then* $y \precsim^{\mathrm{N}*}_{\mathbf{list}(\tau)} v$ *if and only if one of the following holds:*

- $y = \mathsf{lnil}$ *and* $v = \mathtt{nil}$, *or*
- *There exist morphisms* $z_1 : \mathbf{1} \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau]\!]$, $z_2 : \mathbf{1} \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\mathbf{list}(\tau)]\!]$ *and closed expressions* $e_1 : \tau$, $e_2 : \mathbf{list}(\tau)$ *such that* $y = \mathsf{lcons} \circ \langle z_1, z_2 \rangle$, $v = e_1{::}e_2$, $z_1 \precsim^{\mathrm{N}}_{\tau} e_1$, *and* $z_2 \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} e_2$.

Proof.

$\Rightarrow$: Suppose that $y \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} v$. If $y = \mathsf{lnil}$, then

$$\mathcal{K}_{\mathrm{L}} z[\![v : \mathbf{list}(\tau)]\!] \geq \mathsf{Llist}\,(!) \circ \mathsf{lnil} = \mathsf{lnil}$$

The only value v that satisfies the equation is $v = \mathtt{nil}$, fulfilling the lemma.

Otherwise there must exist morphisms $z_1 : \mathbf{1} \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\tau]\!]$ and $z_2 : \mathbf{1} \to L\mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\mathbf{list}(\tau)]\!]$ such that $y = \mathsf{lcons} \circ \langle z_1, z_2 \rangle$. Because

$$\mathcal{K}_{\mathrm{L}} z[\![v : \mathbf{list}(\tau)]\!] \geq \mathsf{Llist}\,(!) \circ y = \mathsf{lcons} \circ \langle !, \mathsf{Llist}\,(!) \circ z_2 \rangle$$

there must exist closed expressions $e_1 : \tau$ and $e_2 : \mathbf{list}(\tau)$ such that $v = e_1{::}e_2$. We therefore only need to show that $z_1 \precsim^{\mathrm{N}}_{\tau} e_1$ and $z_2 \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} e_2$.

For z_1,

$$z_1 = \mathsf{down} \circ \mathsf{up} \circ z_1 = \mathsf{down} \circ \mathsf{lhd} \circ \mathsf{lcons} \circ \langle z_1, z_2 \rangle = \mathsf{down} \circ \mathsf{lhd} \circ y \precsim^{\mathrm{N}}_{\tau} \mathtt{head}(v)$$

Suppose there exists a value v' such that $\mathtt{head}(v) \Rightarrow_{\mathrm{n}} v'$. The only possible derivation of $\mathtt{head}(v) \Rightarrow_{\mathrm{n}} v'$ is

$$\dfrac{\mathtt{head} \Rightarrow_{\mathrm{n}} \mathtt{head} \qquad \dfrac{e_1{::}e_2 \Rightarrow_{\mathrm{n}} e_1{::}e_2 \quad e_1 \Rightarrow_{\mathrm{n}} v'}{\mathrm{apply}\,(\mathtt{head}, e_1{::}e_2) \Rightarrow v'}}{\mathtt{head}(v) \Rightarrow_{\mathrm{n}} v'}$$

Therefore $e_1 \Rightarrow_{\mathrm{n}} v'$, so $z_1 \precsim^{\mathrm{N}}_{\tau} e_1$.

For z_2,

$$\begin{aligned} \mathsf{up} \circ \mathsf{lcons} \circ \langle !, \mathsf{Llist}\,(!) \circ z_2 \rangle &\leq \mathsf{up} \circ \mathcal{K}_{\mathrm{L}} z[\![v : \mathbf{list}(\tau)]\!] \\ &= \mathsf{up} \circ \mathsf{lcons} \circ \langle !, z' \rangle \end{aligned}$$

where $z' = \mathsf{up} \circ \mathcal{K}_{\mathrm{L}} z[\![v_2 : \mathbf{list}(\tau)]\!]$ if there exists a v_2 such that $e_2 \Rightarrow_{\mathrm{n}} v_2$ and $z' = \bot$ otherwise. Furthermore, by the pointwise ordering of lists, $\mathsf{Llist}\,(!) \circ z_2 \leq z'$.

If $z_2 = \bot$, then $z_2 \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} e_2$. Otherwise there exists a morphism $y_2 : \mathbf{1} \to \mathcal{T}_{\mathrm{E}}^{\mathrm{N}}[\![\mathbf{list}(\tau)]\!]$ such that $z_2 = \mathsf{up} \circ y_2$. Thus $\mathsf{Llist}\,(!) \circ z_2 = \mathsf{up} \circ ! \leq z'$, so $z' \neq \bot$. Therefore there exists a value v_2 such that $e_2 \Rightarrow_{\mathrm{n}} v_2$. We immediately know that $\mathsf{Llist}\,! \circ y_2 \leq \mathsf{up} \circ z' = \mathsf{up} \circ \mathcal{K}_{\mathrm{L}} z[\![v_2 : \mathbf{list}(\tau)]\!]$. What is left to show is that for all deconstructors $p_{\mathrm{L}^z_k}$, $p_{\mathrm{L}^z_k} \circ y_2 \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} [v_2/x] e_{\mathrm{L}^z_k}$.

Let $k \geq 1$. Then

$$\begin{array}{rcl} \mathsf{down} \circ p_{\mathrm{L}^Z_k} \circ y_2 & = & (\mathsf{down} \circ p_{\mathrm{L}^Z_k})^{\perp} \circ \mathsf{ltl} \circ \mathsf{lcons} \circ \langle z_1, z_2 \rangle \\ & = & \mathsf{down} \circ p_{\mathrm{L}^Z_{k+1}} \circ \mathsf{lcons} \circ \langle z_1, z_2 \rangle \\ & = & \mathsf{down} \circ p_{\mathrm{L}^Z_{k+1}} \circ y \\ & \precsim^{\mathrm{N}}_{\tau} & [e_1{::}e_2/x]e_{\mathrm{L}^Z_{k+1}} \\ & \equiv & e_{\mathrm{L}^Z_k}(\mathtt{tail}(e_1{::}e_2)) \end{array}$$

Suppose there exists a value v such that $e_{\mathrm{L}^Z_k}(\mathtt{tail}(e_1{::}e_2)) \Rightarrow_{\mathrm{n}} v$, i.e., $\mathtt{head}(\mathtt{tail}^{(k)}(e_1{::}e_2)) \Rightarrow_{\mathrm{n}} v$. The derivation of this evaluation must include the rule

$$\frac{\mathtt{tail}^{(k)}(e_1{::}e_2) \Rightarrow_{\mathrm{n}} e_1'{::}e_2' \quad e_1' \Rightarrow_{\mathrm{n}} v}{\mathrm{napply}(\mathtt{head}, \mathtt{tail}^{(k)}(e_1{::}e_2)) \Rightarrow v}$$

Thus, by Lemma B.2.4, $\mathtt{tail}^{(k-1)}(e_2) \Rightarrow_{\mathrm{n}} e_1'{::}e_2'$, and by Lemma B.2.5 $\mathtt{tail}^{(k-1)}(v_2) \Rightarrow_{\mathrm{n}} e_1'{::}e_2'$. Thus via the derivation

$$\frac{\mathtt{tail}^{(k-1)}(v_2) \Rightarrow_{\mathrm{n}} e_1'{::}e_2' \quad e_1' \overset{t_2}{\Rightarrow}_{\mathrm{n}} v}{\mathrm{napply}(\mathtt{head}, \mathtt{tail}^{(k-1)}(v_2)) \Rightarrow v}$$

we know that $\mathtt{head}(\mathtt{tail}^{(k-1)}(v_2)) \Rightarrow_{\mathrm{n}} v$, so

$$p_{\mathrm{L}^Z_k} \circ y_2 \precsim^{\mathrm{N}}_{\tau} \mathtt{head}(\mathtt{tail}^{(k-1)}(v_2)) \equiv [v_2/x]e_{\mathrm{L}^Z_k}$$

Thus $y_2 \precsim^{\mathrm{N}^*}_{\mathbf{list}(\tau)} v_2$.

Case $\Leftarrow$: Suppose that the properties of the lemma hold. If $y = \mathsf{lnil}$ and $v = \mathtt{nil}$, then

$$\mathsf{Llist}\,(!) \circ y = \mathsf{lnil} = \mathcal{K}_{\mathrm{L}^Z}[\![\mathtt{nil} : \mathbf{list}(\tau)]\!]$$

Furthermore, for all $k \geq 1$,

$$\mathsf{down} \circ p_{\mathrm{L}^Z_k} \circ y = \mathsf{down} \circ p_{\mathrm{L}^Z_k} \circ \mathsf{lnil} = \perp \precsim^{\mathrm{N}}_{\tau} [v/x]e_{\mathrm{L}^Z_k}$$

Thus $y \precsim^{\mathrm{N}^*}_{\mathbf{list}(\tau)} v$.

Otherwise there exists morphisms $z_1 : \mathbf{1} \rightarrow L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau]\!]$, $z_2 : \mathbf{1} \rightarrow L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\mathbf{list}(\tau)]\!]$ and closed expression $e_1 : \tau_1$, $e_2 : \tau_2$ such that $y = \mathsf{lcons} \circ \langle z_1, z_2 \rangle$, $v = e_1{::}e_2$, $z_1 \precsim^{\mathrm{N}}_{\tau} e_1$ and $z_2 \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} e_2$. For structure,

$$\begin{array}{rcl} \mathsf{Llist}\,(!) \circ y & = & \mathsf{Llist}\,(!) \circ \mathsf{lcons} \circ \langle z_1, z_2 \rangle \\ & = & \mathsf{lcons} \circ \langle !, \mathsf{Llist}\,(!) \circ z_2 \rangle \end{array}$$

If $z_2 = \perp$, then

$$\mathsf{lcons} \circ \langle !, L(\mathsf{Llist}\,(!)) \circ z_2 \rangle = \mathsf{lcons} \circ \langle !, \perp \rangle \leq \mathcal{K}_{\mathrm{L}^Z}[\![v : \mathbf{list}(\tau)]\!]$$

Otherwise, as $z_2 \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} e_2$, there exists a value v_2 and a morphism y_2 such that $e_2 \Rightarrow_{\mathrm{n}} v_2$, $z_2 = \mathsf{up} \circ y_2$, and $y_2 \precsim^{\mathrm{N}^*}_{\mathbf{list}(\tau)} v_2$. Therefore

$$L(\mathsf{Llist}\,(!)) \circ z_2 = \mathsf{up} \circ \mathsf{Llist}\,(!) \circ y_2 \leq \mathsf{up} \circ \mathcal{K}_{\mathrm{L}^Z}[\![v_2 : \mathbf{list}(\tau)]\!]$$

Thus by the ordering on lists,

$$\begin{array}{rcl} \mathsf{Llist}\,(!) \circ y & \leq & \mathsf{lcons} \circ \langle !, \mathsf{up} \circ \mathcal{K}_{\mathrm{L}^Z}[\![v_2 : \mathbf{list}\,(\tau)]\!]\rangle \\ & = & \mathcal{K}_{\mathrm{L}^Z}[\![v : \mathbf{list}\,(\tau)]\!] \end{array}$$

For deconstructors,

$$\begin{array}{rcl} \mathsf{down} \circ p_{\mathrm{L}^Z_1} \circ y & = & \mathsf{down} \circ \mathsf{lhd} \circ \mathsf{lcons} \circ \langle z_1, z_2\rangle \\ & = & \mathsf{down} \circ \mathsf{up} z_1 \\ & = & z_1 \\ & \precsim^{\mathrm{N}}_\tau & e_1 \end{array}$$

Suppose there exists a value v_1 such that $e_1 \Rightarrow_{\mathrm{n}} v_1$. Given the derivation

$$\frac{e_1{::}e_2 \Rightarrow_{\mathrm{n}} e_1{::}e_2 \quad e_1 \Rightarrow_{\mathrm{n}} v_1}{\mathrm{napply}\,(\mathtt{head}, e_1{::}e_2) \Rightarrow v_1}$$

we know that $[v/x]e_{\mathrm{L}^Z_1} \equiv \mathtt{head}(e_1{::}e_2) \Rightarrow_{\mathrm{n}} v_1$ as well. Therefore $\mathsf{down} \circ p_{\mathrm{L}^Z_1} \circ y \precsim^{\mathrm{N}}_\tau [v/x]e_{\mathrm{L}^Z_1}$.

For $k > 1$, we know that by monad properties $\mathsf{down} \circ g^\perp = (\mathsf{down} \circ g)^\perp$. Therefore

$$\begin{array}{rcl} \mathsf{down} \circ p_{\mathrm{L}^Z_k} \circ y & = & \mathsf{down} \circ p_{\mathrm{L}^Z_{k-1}}{}^\perp \circ \mathsf{ltl} \circ \mathsf{lcons} \circ \langle z_1, z_2\rangle \\ & = & \mathsf{down} \circ p_{\mathrm{L}^Z_{k-1}}{}^\perp \circ \mathsf{ltl} \circ \mathsf{lcons} \circ \langle z_1, z_2\rangle \\ & = & (\mathsf{down} \circ p_{\mathrm{L}^Z_{k-1}})^\perp \circ z_2 \end{array}$$

If $z_2 = \perp$ then

$$\mathsf{down} \circ p_{\mathrm{L}^Z_k} \circ y = (\mathsf{down} \circ p_{\mathrm{L}^Z_{k-1}})^\perp \circ z_2 = \perp \precsim^{\mathrm{N}}_\tau [e_1{::}e_2/x]e_{\mathrm{L}^Z_k}$$

If $z_2 \neq \perp$ then there exist a value v_2 and a morphism y_2 such that $e_2 \Rightarrow_{\mathrm{n}} v_2$, $z_2 = \mathsf{up} \circ y_2$, and $y_2 \precsim^{\mathrm{N}^*}_{\mathbf{list}(\tau)} v_2$. Therefore

$$\begin{array}{rcl} \mathsf{down} \circ p_{\mathrm{L}^Z_k} \circ y & = & (\mathsf{down} \circ p_{\mathrm{L}^Z_{k-1}})^\perp \circ z_2 \\ & = & \mathsf{down} \circ p_{\mathrm{L}^Z_{k-1}} \circ y_2 \\ & \precsim^{\mathrm{N}}_\tau & [v_2/x]e_{\mathrm{L}^Z_{k-1}} \\ & \equiv & \mathtt{head}(\mathtt{tail}^{(k-2)}(v_2)) \end{array}$$

Suppose that $\mathtt{head}(\mathtt{tail}^{(k-2)}(v_2)) \Rightarrow_{\mathrm{n}} v_2$ for some value v_2. Its derivation then must include the following:

$$\frac{\mathtt{tail}^{(k-2)}(v_2) \Rightarrow_{\mathrm{n}} e_1'{::}e_2' \quad e_1' \Rightarrow_{\mathrm{n}} v'}{\mathrm{napply}\,(\mathtt{head}, \mathtt{tail}^{(k-2)}(v_2)) \Rightarrow v'}$$

for some closed expressions e_1' and e_2'. By Lemma B.2.5 $\mathtt{tail}^{(k-2)}(e_2) \Rightarrow_{\mathrm{n}} e_1'{::}e_2'$ and by Lemma B.2.4 $\mathtt{tail}^{(k-1)}(v) \Rightarrow_{\mathrm{n}} e_1'{::}e_2'$ so with the rule

$$\frac{\mathtt{tail}^{(k-1)}(v) \Rightarrow_{\mathrm{n}} e_1'{::}e_2' \quad e_1' \Rightarrow_{\mathrm{n}} v_2}{\mathrm{napply}\,(\mathtt{head}, \mathtt{tail}^{(k-1)}(v)) \Rightarrow v_2}$$

we know that $[v/x]e_{\mathrm{L}^Z_k} \Rightarrow_{\mathrm{n}} v$. Therefore $p_{\mathrm{L}^Z_k} \circ y \precsim^{\mathrm{N}}_\tau [v/x]e_{\mathrm{L}^Z_k}$, which means that $y \precsim^{\mathrm{N}^*}_{\mathbf{list}(\tau)} v$.

□

With the above theorem, it is simple to show that the four list constants are adequate.

`nil`:

By the second rule of Theorem B.2.6,

$$\mathcal{C}^{\mathrm{N}}_{\mathrm{E}}[\![\texttt{nil}]\!] = \mathsf{up} \circ \mathsf{lnil} \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} \texttt{nil}$$

so `nil` is adequate.

`cons`: Let $z_1 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\tau]\!]$ and e_1 be a closed expression of type τ such that $z_1 \precsim^{\mathrm{N}}_{\tau} e_1$. Similarly, let $z_2 : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\mathbf{list}(\tau)]\!]$ and e_2 be a closed expression of type $\mathbf{list}(\tau)$ such that $z_2 \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} e_2$. Then by the third rule of Theorem B.2.6,

$$\mathcal{C}^{\mathrm{N}}_{\perp}[\![\mathsf{cons}]\!] \circ \langle\rangle(z_1, z_2) = \mathsf{up} \circ \mathsf{lcons} \circ \langle z_1, z_2\rangle \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} e_1{::}e_2$$

so `cons` is adequate.

`head`: Let $z : \mathbf{1} \to L\mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\mathbf{list}(\tau)]\!]$ and e be a closed expression of type $\mathbf{list}(\tau)$ such that $z \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} e$. If $z = \perp$ or $z = \mathsf{up} \circ \mathsf{lnil}$ then

$$\mathcal{C}^{\mathrm{N}}_{\perp}[\![\texttt{head}]\!] \circ \langle\rangle(z) = (\mathsf{down} \circ \mathsf{lhd})^{\perp} \circ z = \perp \precsim^{\mathrm{N}}_{\tau} \texttt{head}(e)$$

Therefore assume that for some z_1, z_2, $z = \mathsf{up} \circ \mathsf{lcons} \circ \langle z_1, z_2\rangle$. This means that there exist a value v and a morphism $y : \mathbf{1} \to \mathcal{T}^{\mathrm{N}}_{\mathrm{E}}[\![\mathbf{list}(\tau)]\!]$ such that $e \Rightarrow_{\mathrm{n}} v$, $z = \mathsf{up} \circ y$, and $y \precsim^{\mathrm{N}*}_{\mathbf{list}(\tau)} v$. Therefore there must exist expressions e_1 and e_2 such that $v = e_1{::}e_2$, $z_1 \precsim^{\mathrm{N}}_{\tau} e_1$, and $z_2 \precsim^{\mathrm{N}}_{\mathbf{list}(\tau)} e_2$. Therefore

$$\begin{aligned}
\mathcal{C}^{\mathrm{N}}_{\perp}[\![\texttt{head}]\!] \circ \langle\rangle(z) &= (\mathsf{down} \circ \mathsf{lhd})^{\perp} \circ \mathsf{up} \circ \mathsf{lcons} \circ \langle z_1, z_2\rangle \\
&= \mathsf{down} \circ \mathsf{lhd} \circ \mathsf{lcons} \circ \langle z_1, z_2\rangle \\
&= \mathsf{down} \circ \mathsf{up} \circ z_1 \\
&= z_1 \\
&\precsim^{\mathrm{N}}_{\tau} e_1
\end{aligned}$$

Suppose $e_2 \Rightarrow_{\mathrm{n}} v_1$. Then via the rule

$$\frac{e \Rightarrow_{\mathrm{n}} e_1{::}e_2 \quad e_1 \Rightarrow_{\mathrm{n}} v_1}{\mathrm{napply}(\texttt{head}, e) \Rightarrow v_1}$$

$\texttt{head}(e) \Rightarrow_{\mathrm{n}} v_1$ as well. Therefore $\mathcal{C}^{\mathrm{N}}_{\perp}[\![\texttt{head}]\!] \circ \langle\rangle(z) \precsim^{\mathrm{N}}_{\tau} \texttt{head}(e)$, so `head` is adequate.

`tail`: Due to Theorem B.2.6, the proof is similar to the one above.

`nil?`: The proof that this is adequate is very similar to the call-by-value proof.

Appendix C

The Accept program: Theorem 5.1.5

Remember from Chapter **??** that $\mathsf{kkstrof}(k, s)$ is the value of applying k to s except that $\perp$ is converted to $\mathbf{tt}$. Also, $\mathsf{kkstr}(k, s)$ is the continuation that is false (with zero cost) such that for all strings s' except s, where it returns the value of $\mathsf{kkstrof}(k, s)$.

Theorem 5.1.5 *For any continuation $k : \mathbf{1} \to \mathcal{T}^{\mathrm{V}}[\![\mathbf{cont}]\!]$, any regular expression r, and any string s, let $[s_1, \ldots, s_n] = S_{\text{cont}}(r, s)$, and let i be the smallest integer $1 \leq i \leq n$ where k is not false on s, or n if no such integer exists. Then one of the following two possibilities hold:*

- *either $n = 0$ and there exists a cost t such that for all continuations k',*

$$\mathsf{acc}(r, k', s) = [\![t]\!] \circ \eta \circ \mathsf{ff}$$

- *or $n > 0$ and*

$$\mathsf{acc}(r, k, s) = [\![t + \sum_{j=1}^{i-1} t_j]\!] \circ \mathbf{apply}(\eta \circ k, s_i)$$

where $\mathsf{acc}(r, \mathsf{kkstr}(k, s_i), s) = [\![t]\!] \circ \eta \circ \mathsf{kkstrof}(k, s_i)$*, and, for* $1 \leq j < i$*,*

$$\mathbf{apply}(\eta \circ k, s_j) = [\![t_j]\!] \circ \eta \circ \mathsf{ff}$$

Proof. By induction on the structure of r and the length of s. There are a number of cases to check; however, we can simplify them somewhat by letting

$$\mathsf{kap}([s_1, \ldots, s_n])(k, i) = \mathbf{apply}(\eta \circ k, s_i)$$

when $n > 0$ (and thus $1 \leq i \leq n$), and

$$\mathsf{kap}([\,])(k, i) = \eta \circ \mathsf{ff}$$

when $n = 0$ (and thus $i = 0$ as well). Then, the theorem implies that

$$\mathsf{acc}(r, k, s) = [\![t + \sum_{j=1}^{i-1} t_j]\!] \circ \mathsf{kap}(S_{\text{cont}}(r, s))(k, i)$$

even when n and thus i are 0. The converse is not necessarily true, as the theorem states a more general result when $n = 0$.

We divide the proof first into cases depending on the structure of r. Within each case we will generally divide into additional cases. The sub-cases vary depending on which type of regular expression we are examining: for example, when $r = a$, the proof depends on the structure of s, but when $r = r_1 r_2$, the result depends on the lengths of the lists $S_{\text{cont}}(r_1, s)$ and $S_{\text{cont}}(r_1 r_2, s)$.

Case $r = a$**:** We have three cases to check, depending on the structure of s:

- $s = \mathsf{snil}(m)$: By definition, $S_{\text{cont}}(r, s) = []$, so $n = 0$ and we must show that the first result holds. For any continuation k',

$$\begin{aligned}&\mathsf{acc}(a, k', \mathsf{snil}(m))\\ &\quad = [\![t_{\text{rcase}} + t_{\text{app}}]\!] \circ \mathbf{apply}(\mathsf{accchar}(k', \mathsf{snil}(m)), a)\\ &\quad = [\![t_{\text{true}} + t_{\text{fst}} + t_{\text{nil?}} + t_{\text{rcase}} + 3t_{\text{app}}]\!] \circ \eta \circ \mathsf{ff}\end{aligned}$$

  which satisfies the theorem with $t = t_{\text{true}} + t_{\text{fst}} + t_{\text{nil?}} + t_{\text{rcase}} + 3t_{\text{app}}$.

- $s = \mathsf{scons}(a', s')$, $a \neq a'$: This case is similar to the previous case in that for all k', $\mathsf{acc}(r, k', s)$ returns false with a constant time.

- $s = \mathsf{scons}(a, s')$: Then $S_{\text{cont}}(r, s) = [s']$, so we must show that the second part of the theorem holds with $i = 1$ (as no other value for i is possible). First, for any continuation k',

$$\begin{aligned}&\mathsf{acc}(a, k', \mathsf{scons}(a, s'))\\ &\quad = [\![t_{\text{rcase}} + t_{\text{app}}]\!] \circ [\![t_{\text{false}} + 3t_{\text{app}} + 2t_{\text{fst}} + t_{\text{nil?}} + t_{\text{head}}]\!]\\ &\qquad\quad \circ [\![t_{\text{app}} + t_{\text{fst}} + t_{+} + t_{\text{snd}} + t_{\text{tail}} + t_{\text{true}} + t_{=}]\!] \circ \mathbf{apply}(\eta \circ k', s')\\ &\quad = [\![t_{+} + t_{\text{snd}} + t_{\text{tail}} + t_{\text{true}} + t_{=} + t_{\text{false}} + 3t_{\text{fst}} + t_{\text{nil?}} + t_{\text{head}} + t_{\text{rcase}} + 5t_{\text{app}}]\!]\\ &\qquad\quad \circ \mathbf{apply}(\eta \circ k', s')\end{aligned}$$

  Thus let $t = t_{+} + t_{\text{snd}} + t_{\text{tail}} + t_{\text{true}} + t_{=} + t_{\text{false}} + 3t_{\text{fst}} + t_{\text{nil?}} + t_{\text{head}} + t_{\text{rcase}} + 5t_{\text{app}}$. Then

$$\mathsf{acc}(r, k, s) = [\![t]\!] \circ \mathbf{apply}(\eta \circ k, s')$$

  where

$$\mathsf{acc}(r, \mathsf{kkstr}(k, s'), s) = [\![t]\!] \circ \mathbf{apply}(\eta \circ \mathsf{kkstr}(k, s'), s') = [\![t]\!] \circ \eta \circ \mathsf{kkstrof}(k, s')$$

  Because $i = 1$ and $s' = s_i$, we thus know that the theorem holds.

Case $r = r_1|r_2$**:** Let $[s_1, \ldots, s_n] = S_{\text{cont}}(r_1, s)$ and $[s'_1, \ldots, s'_m] = S_{\text{cont}}(r_2, s)$. Then

$$S_{\text{cont}}(r_1|r_2, s) = [s_1, \ldots, s_n, s'_1, \ldots, s'_m]$$

Let $s''_j = s_j$ for $1 \leq j \leq n$ and $s''_j = s'_{j-n}$ for $n < j \leq n + m$. Let i be the smallest integer such that $1 \leq i \leq n + m$ and k does not return false on s''_i, or $n + m$ if no such integer exists. We have three cases that depend on the size of n and m and on the value of i relative to n and m:

- $n + m = 0$: Then n and m are both 0, so by the induction hypothesis for any continuation k' there exist costs t_1 and t_2 such that

$$\mathsf{acc}(r_1, k', s) = [\![t_1]\!] \circ \eta \circ \mathsf{ff} \quad \text{and} \quad \mathsf{acc}(r_2, k', s) = [\![t_2]\!] \circ \eta \circ \mathsf{ff}$$

  Therefore

$$\begin{aligned}&\mathsf{acc}(r_1|r_2, k', s)\\ &\quad = [\![t_{\text{rcase}} + t_{\text{app}}]\!] \circ \mathbf{apply}(\mathsf{accor}(k', s), r_1, r_2)\\ &\quad = [\![t_{\text{rcase}} + 5t_{\text{app}} + t_{\text{rec}}]\!] \circ \ \mathbf{cond}([\![t_1]\!] \circ \eta \circ \mathsf{ff})\ \mathbf{of}\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{True:}\quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt}\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{False:}\quad [\![t_{\text{false}} + 2t_{\text{app}} + t_{\text{rec}}]\!] \circ \mathsf{acc}(r_2, k, s)\\ &\quad = [\![t_1 + t_{\text{rcase}} + 5t_{\text{app}} + t_{\text{rec}}]\!] \circ [\![t_{\text{false}}]\!] \circ [\![t_2]\!] \circ \eta \circ \mathsf{ff}\\ &\quad = [\![t_2 + t_{\text{false}} + t_1 + t_{\text{rcase}} + 7t_{\text{app}} + 2t_{\text{rec}}]\!] \circ \eta \circ \mathsf{ff}\end{aligned}$$

  so the theorem holds with $t = t_2 + t_{\text{false}} + t_1 + t_{\text{rcase}} + 7t_{\text{app}} + 2t_{\text{rec}}$.

- $n+m>0$ and $i \leq n$: Then $n>0$, so by the induction hypothesis

$$\mathsf{acc}(r_1, k, s) = [\![t_1 + \sum_{j=1}^{i-1} t'_j]\!] \circ \mathbf{apply}(\eta \circ k, s_i)$$

where

$$\mathsf{acc}(r_1, \mathsf{kkstr}(k, s_i), s) = [\![t_1]\!] \circ \eta \circ \mathsf{kkstrof}(k, s_i)$$

and, for $1 \leq j < i$,

$$\mathbf{apply}(\eta \circ k, s_j) = [\![t'_j]\!] \circ \eta \circ \mathsf{ff}$$

Suppose that, k is false on s_i, i.e., for some cost t'_i, $\mathbf{apply}(\eta \circ k, s_i) = [\![t'_i]\!] \circ \eta \circ \mathsf{ff}$. Then $i = n = n+m$, which means that $m = 0$. In that case there exists a cost t_2 such that

$$\mathsf{acc}(r_2, k, s) = [\![t_2]\!] \circ \eta \circ \mathsf{ff}$$

Therefore,

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{acc}(r_1|r_2, \mathsf{kkstr}(k, s_i), s)} \\
= & [\![t_{\mathrm{rcase}} + 5t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ & \mathbf{cond}\,([\![t_1]\!] \circ \eta \circ \mathsf{ff})\ \mathbf{of} \\
& & \quad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
& & \quad \mathbf{False:} \quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ [\![t_2]\!] \circ \eta \circ \mathsf{ff} \\
= & \multicolumn{2}{l}{[\![t_2 + t_{\mathrm{false}} + t_1 + t_{\mathrm{rcase}} + 7t_{\mathrm{app}} + 2t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{ff}}
\end{array}$$

so let $t = t_2 + t_{\mathrm{false}} + t_1 + t_{\mathrm{rcase}} + 7t_{\mathrm{app}} + 2t_{\mathrm{rec}}$. Then

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{acc}(r_1|r_2, k, s)} \\
= & [\![t_{\mathrm{rcase}} + 5t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ & \mathbf{cond}\,([\![t_1 + \sum_{j=1}^{i-1} t'_j]\!] \circ [\![t'_i]\!] \circ \eta \circ \mathsf{ff})\ \mathbf{of} \\
& & \quad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
& & \quad \mathbf{False:} \quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ [\![t_2]\!] \circ \eta \circ \mathsf{ff} \\
= & \multicolumn{2}{l}{[\![t_2 + t_{\mathrm{false}} + t_{\mathrm{rcase}} + 7t_{\mathrm{app}} + 2t_{\mathrm{rec}} + t_1 + \sum_{j=1}^{i} t'_j]\!] \circ \eta \circ \mathsf{ff}} \\
= & \multicolumn{2}{l}{[\![t + \sum_{j=1}^{i-1} t'_j]\!] \circ [\![t'_i]\!] \circ \eta \circ \mathsf{ff}} \\
= & \multicolumn{2}{l}{[\![t + \sum_{j=1}^{i-1} t'_j]\!] \circ \mathbf{apply}(\eta \circ k, s_i)}
\end{array}$$

If k is not false on s_i, then $\mathsf{kkstrof}(k, s_i) = \mathsf{tt}$, so

$$\mathsf{acc}(r_1, \mathsf{kkstr}(k, s_i), s) = [\![t_1]\!] \circ \eta \circ \mathsf{tt}$$

and thus

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{acc}(r_1|r_2, \mathsf{kkstr}(k, s_i), s)} \\
= & [\![t_{\mathrm{rcase}} + 5t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ & \mathbf{cond}\,([\![t_1]\!] \circ \eta \circ \mathsf{tt})\ \mathbf{of} \\
& & \quad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
& & \quad \mathbf{False:} \quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ \mathsf{acc}(r_2, \mathsf{kkstr}(k, s_i), s) \\
= & \multicolumn{2}{l}{[\![t_{\mathrm{true}} + t_1 + t_{\mathrm{rcase}} + 5t_{\mathrm{app}} + t_{\mathrm{rec}}]\!] \circ \eta \circ \mathsf{tt}}
\end{array}$$

Therefore let $t = t_{\mathrm{true}} + t_1 + t_{\mathrm{rcase}} + 5t_{\mathrm{app}} + t_{\mathrm{rec}}$. Now, we know that k is not false on s_i, which means that either $\mathbf{apply}(\eta \circ k, s_i) = \perp$ or, for some t', $\mathbf{apply}(\eta \circ k, s_i) = [\![t']\!] \circ \eta \circ \mathsf{tt}$.

If the former, then

$$\begin{array}{rl}\mathsf{acc}(r_1|r_2,k,s) & \\ = & [\![t_{\rm rcase}+5t_{\rm app}+t_{\rm rec}]\!]\circ\ \ \mathbf{cond}(\bot)\ \mathbf{of} \\ & \qquad\qquad \mathbf{True:}\quad [\![t_{\rm true}]\!]\circ\eta\circ\mathsf{tt} \\ & \qquad\qquad \mathbf{False:}\quad [\![t_{\rm false}]\!]\circ\mathsf{acc}(r_2,k,s) \\ = & \bot \\ = & [\![t+\sum_{j=1}^{i-1}t'_j]\!]\circ\mathbf{apply}(\eta\circ k,s_i)\end{array}$$

Otherwise,

$$\begin{array}{rl}\mathsf{acc}(r_1|r_2,k,s) & \\ = & [\![t_{\rm rcase}+5t_{\rm app}+t_{\rm rec}]\!]\circ\ \ \mathbf{cond}([\![t_1+\sum_{j=1}^{i-1}t'_j]\!]\circ[\![t']\!]\circ\eta\circ\mathsf{tt})\ \mathbf{of} \\ & \qquad\qquad \mathbf{True:}\quad [\![t_{\rm true}]\!]\circ\eta\circ\mathsf{tt} \\ & \qquad\qquad \mathbf{False:}\quad [\![2t_{\rm app}+t_{\rm rec}+t_{\rm false}]\!]\circ\mathsf{acc}(r_2,k,s) \\ = & [\![t_{\rm true}+t'+t_1+\sum_{j=1}^{i-1}t'_j+t_{\rm rcase}+5t_{\rm app}+t_{\rm rec}]\!]\circ\eta\circ\mathsf{tt} \\ = & [\![t+\sum_{j=1}^{i-1}t'_j]\!]\circ[\![t']\!]\circ\eta\circ\mathsf{tt} \\ = & [\![t+\sum_{j=1}^{i-1}t'_j]\!]\circ\mathbf{apply}(\eta\circ k,s_i)\end{array}$$

- $n+m>0$ and $i>n$: Then k is false on all s_j, $1\le j\le n$, so by the induction hypothesis,

$$\mathsf{acc}(r_1,k,s)=[\![t_1+\sum_{j=1}^{n-1}t'_j]\!]\circ\mathsf{kap}([s_1,\ldots,s_n])(k,n)$$

where $\mathsf{acc}(r_1,\mathsf{kkstr}(k,s_n),s)=[\![t_1]\!]\circ\eta\circ\mathsf{kkstrof}(k,s_n)=[\![t_1]\!]\circ\eta\circ\mathsf{ff}$. Because k is false on $[s_1,\ldots,s_n]$, we know that there exists a cost t'_n such that $\mathsf{kap}([s_1,\ldots,s_n])(k,n)$ equals $[\![t'_n]\!]\circ\eta\circ\mathsf{ff}$. Therefore

$$\mathsf{acc}(r_1,k,s)=[\![t_1+\sum_{j=1}^{n}t'_j]\!]\circ\eta\circ\mathsf{ff}$$

Also, because $\mathsf{kkstr}(k,s''_i)$ has the same values on $[s_1,\ldots,s_n]$ as $\mathsf{kkstr}(k,s_n)$,

$$\mathsf{acc}(r_1,\mathsf{kkstr}(k,s''_i),s)=[\![t_1]\!]\circ\eta\circ\mathsf{ff}$$

as well. Lastly, by the induction hypothesis,

$$\mathsf{acc}(r_2,k,s)=[\![t_2+\sum_{j=1}^{i-n-1}t''_j]\!]\circ\mathbf{apply}(\eta\circ k,s'_{i-n})$$

where $\mathsf{acc}(r_2,\mathsf{kkstr}(k,s'_{i-n}),s)=[\![t_2]\!]\circ\eta\circ\mathsf{kkstrof}(k,s'_{i-n})$ and, for $1\le j<i-n$,

$$\mathbf{apply}(\eta\circ k,s'_j)=[\![t''_j]\!]\circ\eta\circ\mathsf{ff}$$

For $n<j<i$, let $t'_j=t''_{j-n}$. Then, as $s''_i=s'_{i-n}$,

$$\begin{array}{rl}\mathsf{acc}(r_1|r_2,\mathsf{kkstr}(k,s''_i),s) & \\ = & [\![t_{\rm rcase}+5t_{\rm app}+t_{\rm rec}]\!] \\ & \quad\circ\ \mathbf{cond}([\![t_1]\!]\circ\eta\circ\mathsf{ff})\ \mathbf{of} \\ & \qquad \mathbf{True:}\quad [\![t_{\rm true}]\!]\circ\eta\circ\mathsf{tt} \\ & \qquad \mathbf{False:}\quad [\![2t_{\rm app}+t_{\rm rec}+t_{\rm false}]\!]\circ[\![t_2]\!]\circ\eta\circ\mathsf{kkstrof}(k,s''_i) \\ = & [\![t_2+t_{\rm false}+t_1+t_{\rm rcase}+7t_{\rm app}+2t_{\rm rec}]\!]\circ\eta\circ\mathsf{kkstrof}(k,s''_i)\end{array}$$

so, if $t = t_2 + t_{\text{false}} + t_1 + t_{\text{rcase}} + 7t_{\text{app}} + 2t_{\text{rec}}$,

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{acc}(r_1|r_2, k, s)} \\
= & [\![t_{\text{rcase}} + 5t_{\text{app}} + t_{\text{rec}}]\!] \\
& \quad \circ\ \mathbf{cond}([\![t_1 + \sum_{j=1}^{n} t'_j]\!] \circ \eta \circ \mathsf{ff})\ \mathbf{of} \\
& \qquad \mathbf{True:} \quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
& \qquad \mathbf{False:} \quad [\![2t_{\text{app}} + t_{\text{rec}} + t_{\text{false}}]\!] \circ [\![t_2 + \sum_{j=n+1}^{i-1} t'_j]\!] \circ \mathbf{apply}(\eta \circ k, s'_{i-n}) \\
= & [\![t + \sum_{j=1}^{i-1} t'_j]\!] \circ \mathbf{apply}(\eta \circ k, s''_i)
\end{array}
$$

As there are no other possible values for m, n, and i, the theorem holds when $r = r_1|r_2$.

Case $r = r_1 r_2$**:** Let $[s_1, \ldots, s_n] = S_{\text{cont}}(r_1, s)$ and $[s'_1, \ldots, s'_m] = S_{\text{cont}}(r_1 r_2, s)$. By the definition of S_{cont}, we know that $[s'_1, \ldots, s'_m]$ can be partitioned as follows:

$$
\overbrace{s'_1 \ldots s'_{i_1}}^{S_{\text{cont}}(r_2, s_1)} \mid \overbrace{s'_{i_1+1} \ldots s'_{i_2}}^{S_{\text{cont}}(r_2, s_2)} \mid \ldots \mid \overbrace{s'_{i_{n-1}+1} \ldots s'_m}^{S_{\text{cont}}(r_2, s_n)}
$$

We need to check three cases based on the values of n and m:

- $n = 0$: Then by the definition of S_{cont}, $m = 0$ as well, and by the induction hypothesis, there exists a cost t' such that, for all continuations k',

$$\mathsf{acc}(r_1, k', s) = [\![t']\!] \circ \eta \circ \mathsf{ff}$$

Therefore, for all continuations k',

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{acc}(r_1 r_2, k', s)} \\
= & [\![t_{\text{rcase}} + t_{\text{app}}]\!] \circ \mathbf{apply}(\mathsf{accseq}(k', s), r_1, r_2) \\
= & [\![t_{\text{rcase}} + 7t_{\text{app}} + 2t_{\text{rec}}]\!] \circ \mathsf{acc}(r_1, \mathsf{acck}(r_2, k'), s) \\
= & [\![t' + t_{\text{rcase}} + 7t_{\text{app}} + 2t_{\text{rec}}]\!] \circ \eta \circ \mathsf{ff}
\end{array}
$$

proving that the theorem holds in this case with $t = t' + t_{\text{rcase}} + 5t_{\text{app}} + 2t_{\text{rec}}$.

- $n > 0$ and $m = 0$: This means that for all $1 \leq j \leq n$, $S_{\text{cont}}(r_2, s_j) = [\,]$. Then by the induction hypothesis for each $1 \leq j \leq n$ there exists a cost t'_j such that for all continuations k'

$$\mathsf{acc}(r_2, k', s_j) = [\![t'_j]\!] \circ \eta \circ \mathsf{ff}$$

This also means that, for all continuations k', $\mathsf{acck}(r_2, k')$ is false on each s_j as well, so $\mathsf{kkstr}(\mathsf{acck}(r_2, k'), s_n)$ is the same morphism for all k'. Therefore, by the induction hypothesis, there exists a cost t_1 such that $\mathsf{acc}(r_1, \mathsf{kkstr}(\mathsf{acck}(r_2, k'), s_n), s) = [\![t_1]\!] \circ \eta \circ \mathsf{ff}$ and

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{acc}(r_1 r_2, \mathsf{kkstr}(k', s_n), s)} \\
= & [\![t_{\text{rcase}} + 7t_{\text{app}} + 2t_{\text{rec}}]\!] \circ \mathsf{acc}(r_1, \mathsf{acck}(r_2, k'), s) \\
= & [\![t_{\text{rcase}} + 7t_{\text{app}} + 2t_{\text{rec}}]\!] \circ [\![t_1 + \sum_{j=1}^{n-1} t'_j]\!] \circ \mathbf{apply}(\eta \circ \mathsf{acck}(r_2, k'), s_n) \\
= & [\![t_1 + t_{\text{rcase}} + 7t_{\text{app}} + 2t_{\text{rec}} + \sum_{j=1}^{n-1} t'_j]\!] \circ \mathsf{acc}(r2, k', s_n) \\
= & [\![t_1 + t_{\text{rcase}} + 7t_{\text{app}} + 2t_{\text{rec}} + \sum_{j=1}^{n} t'_j]\!] \circ \eta \circ \mathsf{ff}
\end{array}
$$

proving that the theorem holds in this case with $t = t_1 + t_{\text{rcase}} + 7t_{\text{app}} + 2t_{\text{rec}} + \sum_{j=1}^{n} t'_j$.

- $n > 0$ and $m > 0$: If we set i_0 to 0 and i_n to m, then we know that there exists an integer j such that $i_{(j-1)} < i \leq i_j$. For $j' < j$, we know that for each $i_{(j'-1)} < h \leq i_{j'}$, there exists a cost t'_h such that $\mathbf{apply}(\eta \circ k, s'_h) = [\![t'_h]\!] \circ \eta \circ \mathsf{ff}$, and therefore by the induction hypothesis,

$$\begin{aligned}\mathsf{acc}(r_2, k, s_{j'}) &= [\![t_{j'} + \textstyle\sum_{h=i_{(j'-1)}+1}^{i_{j'}-1} t'_h]\!] \circ \mathbf{apply}(\eta \circ k, s'_{i_{j'}}) \\ &= [\![t_{j'} + \textstyle\sum_{h=i_{(j'-1)}+1}^{i_{j'}} t'_h]\!] \circ \eta \circ \mathsf{ff}\end{aligned}$$

where $\mathsf{acc}(r_2, \mathsf{kkstr}(k, s'_{i_{j'}}), s_{j'}) = [\![t_{j'}]\!] \circ \eta \circ \mathsf{kkstrof}(k, s'_{i_{j'}})$ which equals $[\![t_{j'}]\!] \circ \eta \circ \mathsf{ff}$. Furthermore, for $i_{(j-1)} < h < i$, there exists a cost t'_h such that $\mathbf{apply}(\eta \circ k, s'_h)$ equals $[\![t'_h]\!] \circ \eta \circ \mathsf{ff}$, and that, by the induction hypothesis,

$$\mathsf{acc}(r_2, k, s_j) = [\![t_j + \sum_{h=i_{(j-1)}+1}^{i-1} t'_h]\!] \circ \mathbf{apply}(\eta \circ k, s'_i)$$

where $\mathsf{acc}(r_2, \mathsf{kkstr}(k, s'_i), s_j) = [\![t_j]\!] \circ \eta \circ \mathsf{kkstrof}(k, s'_i)$. Lastly, we know there exists a cost t_0 such that

$$\mathsf{acc}(r_1, \mathsf{kkstr}(k, s_j), s) = [\![t_0]\!] \circ \eta \circ \mathsf{kkstrof}(k, s_j)$$

Next we need to find the value of $\mathsf{acc}(r_1 r_2, \mathsf{kkstr}(k, s'_i), s)$. By the induction hypothesis and the values given above, we have that

$$\begin{aligned}&\mathsf{acc}(r_1 r_2, \mathsf{kkstr}(k, s'_i), s) \\ &\quad= [\![t_{\mathrm{rcase}} + 7t_{\mathrm{app}} + 2t_{\mathrm{rec}}]\!] \circ \mathsf{acc}(r_1, \mathsf{acck}(r_2, \mathsf{kkstr}(k, s'_i)), s) \\ &\quad= [\![t_{\mathrm{rcase}} + 7t_{\mathrm{app}} + 2t_{\mathrm{rec}}]\!] \circ [\![t_0 + \textstyle\sum_{j'=1}^{j-1} t'_j]\!] \circ \mathbf{apply}(\eta \circ \mathsf{acck}(r_2, \mathsf{kkstr}(k, s'_i)), s_j) \\ &\quad= [\![t_0 + t_{\mathrm{rcase}} + 7t_{\mathrm{app}} + 2t_{\mathrm{rec}} + \textstyle\sum_{j'=1}^{j} t'_j]\!] \circ \eta \circ \mathsf{kkstrof}(k, s'_i)\end{aligned}$$

Therefore let $t = t_0 + t_{\mathrm{rcase}} + 5t_{\mathrm{app}} + t_{\mathrm{rec}} + \sum_{j'=1}^{j} t'_j$. Then

$$\begin{aligned}&\mathsf{acc}(r_1 r_2, k, s) \\ &\quad= [\![t_{\mathrm{rcase}} + 7t_{\mathrm{app}} + 2t_{\mathrm{rec}}]\!] \circ \mathsf{acc}(r_1, \mathsf{acck}(r_2, k), s) \\ &\quad= [\![t_{\mathrm{rcase}} + 7t_{\mathrm{app}} + 2t_{\mathrm{rec}}]\!] \circ [\![t_0 + \textstyle\sum_{j'=1}^{j-1}(t'_j + \sum_{h=i_{(j'-1)}+1}^{i_{j'}} t'_h)]\!] \\ &\qquad\quad \circ \mathbf{apply}(\eta \circ \mathsf{acck}(r_2, k), s_j) \\ &\quad= [\![t_0 + t_{\mathrm{rcase}} + 7t_{\mathrm{app}} + 2t_{\mathrm{rec}} + \textstyle\sum_{j'=1}^{j-1}(t'_j + \sum_{h=i_{(j'-1)}+1}^{i_{j'}} t'_h)]\!] \circ [\![t'_j + \sum_{h=i_{(j-1)}+1}^{i-1}]\!] \\ &\qquad\quad \circ \mathbf{apply}(\eta \circ k, s'_i) \\ &\quad= [\![t_0 + t_{\mathrm{rcase}} + 7t_{\mathrm{app}} + 2t_{\mathrm{rec}} + \textstyle\sum_{j'=1}^{j} t'_j + \sum_{h=1}^{i-1} t'_h]\!] \circ \mathbf{apply}(\eta \circ k, s'_i) \\ &\quad= [\![t + \textstyle\sum_{h=1}^{i-1} t'_h]\!] \circ \mathbf{apply}(\eta \circ k, s'_i)\end{aligned}$$

Case $r = (r')^*$**:** Let $[s_1, \ldots, s_n] = S_{\mathrm{cont}}(r', s)$ and $[s, s'_1, \ldots, s'_m] = S_{\mathrm{cont}}((r')^*, s)$. Then there are several cases depending on the value of i and whether k is false or not on s. Note that there is always an $i \geq 1$ as $S_{\mathrm{cont}}((r')^*)$ always starts with s (because the program always calls the continuation on s when evaluating a repetition).

- $i = 1$ and k is false on s: Then the length of $S_{\mathrm{cont}}((r')^*, s)$ must be one, so m must be 0. This could happen in two ways. First n could be 0. Then, as in the sequential case, we know that by the induction hypothesis there exists a cost t' such that for any continuation k',

$$\mathsf{acc}(r', k', s) = [\![t']\!] \circ \eta \circ \mathsf{ff}$$

Also, as k is false on s, there exists a cost t_0 such that $\mathbf{apply}(\eta \circ k, s) = [\![t_0]\!] \circ \eta \circ \mathsf{ff}$. Therefore $\mathbf{apply}(\eta \circ \mathsf{kkstr}(k,s), s) = \eta \circ \mathsf{ff}$ and

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{acc}((r')^*, \mathsf{kkstr}(k,s), s)} \\
= & [\![3t_{\mathrm{app}} + t_{\mathrm{rcase}}]\!] \\
 & \quad \circ\ \mathbf{cond}(\mathbf{apply}(\eta \circ \mathsf{kkstr}(k,s), s))\ \mathbf{of} \\
 & \qquad \mathbf{True:}\quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
 & \qquad \mathbf{False:}\quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ \mathsf{acc}(r', \mathsf{checkk}(\mathsf{kkstr}(k,s), s, r), s) \\
= & [\![5t_{\mathrm{app}} + t_{\mathrm{rcase}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ [\![t']\!] \circ \eta \circ \mathsf{ff} \\
= & [\![t' + 5t_{\mathrm{app}} + t_{\mathrm{rcase}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ \eta \circ \mathsf{kkstrof}(k,s)
\end{array}
$$

thus, setting t to $t' + 5t_{\mathrm{app}} + t_{\mathrm{rcase}} + t_{\mathrm{rec}} + t_{\mathrm{false}}$,

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{acc}((r')^*, k, s)} \\
= & [\![3t_{\mathrm{app}} + t_{\mathrm{rcase}}]\!] \circ\ \mathbf{cond}([\![t_0]\!] \circ \eta \circ \mathsf{ff})\ \mathbf{of} \\
 & \qquad \mathbf{True:}\quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
 & \qquad \mathbf{False:}\quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ [\![t']\!] \circ \eta \circ \mathsf{ff} \\
= & [\![t' + 5t_{\mathrm{app}} + t_{\mathrm{rcase}} + t_{\mathrm{rec}} + t_{\mathrm{false}} + t_0]\!] \circ \eta \circ \mathsf{ff} \\
= & [\![t]\!] \circ \mathbf{apply}(\eta \circ k, s)
\end{array}
$$

proving that the theorem holds in this case.

Alternatively, if n is not 0, then for all s_j, if $s_j \neq s$, then $S_{\mathrm{cont}}((r')^*, s_j) = []$, so for those cases by the induction hypothesis there exists a cost t'_j such that for any continuation k',

$$\mathsf{acc}((r')^*, k', s_j) = [\![t'_j]\!] \circ \eta \circ \mathsf{ff}$$

In order to evaluate r', however, we need to know the value of $\mathsf{checkk}(k', s_j, r')$ on all s_j, including the ones that equal s. By the above statement, for $s_j \neq s$, and any continuation k'

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{apply}(\eta \circ \mathsf{checkk}(k', s, r'), s_j)} \\
= & [\![2t_{\mathrm{snd}} + t_{\mathrm{app}}]\!] \circ [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}} + t_{\neq}]\!] \circ \mathsf{acc}((r')^*, k', s_j) \\
= & [\![t'_j + t_{\mathrm{rec}} + t_{\mathrm{false}} + t_{\neq} + 2t_{\mathrm{snd}} + 3t_{\mathrm{app}}]\!] \circ \eta \circ \mathsf{ff}
\end{array}
$$

and, for $s_j = s$,

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{apply}(\eta \circ \mathsf{checkk}(k', s, r'), s_j)} \\
= & [\![2t_{\mathrm{snd}} + t_{\mathrm{app}}]\!] \circ [\![t_{\mathrm{true}} + t_{=}]\!] \circ \eta \circ \mathsf{ff} \\
= & [\![t_{\mathrm{true}} + t_{=} + 2t_{\mathrm{snd}} + t_{\mathrm{app}}]\!] \circ \eta \circ \mathsf{ff}
\end{array}
$$

If we set $t''_j = t_{\mathrm{true}} + t_{=} + 2t_{\mathrm{snd}} + t_{\mathrm{app}}$ when $s_j = s$ and $t''_j = t'_j + t_{\mathrm{rec}} + t_{\mathrm{false}} + t_{\neq} + 2t_{\mathrm{snd}} + 3t_{\mathrm{app}}$ otherwise, then, by the induction hypothesis,

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{acc}(r', \mathsf{checkk}(k', s, r'), s)} \\
= & [\![t + \sum_{j=1}^{n-1} t''_j]\!] \circ \mathbf{apply}(\eta \circ \mathsf{checkk}(k, s, r'), s_n) \\
= & [\![t + \sum_{j=1}^{n} t''_j]\!] \circ \eta \circ \mathsf{ff}
\end{array}
$$

where $\mathsf{acc}(r', \mathsf{kkstr}(\mathsf{checkk}(k', s, r'), s_n), s) = [\![t]\!] \circ \eta \circ \mathsf{kkstrof}(\mathsf{checkk}(k', s, r'), s_n)$. Note that for any continuation k', $\mathsf{kkstr}(\mathsf{checkk}(k', s, r'), s_n)$ is the same morphism, so the cost t is independent of the continuation as well.

Finally, as k is false on s, there exists a cost t_0 such that $\mathbf{apply}(\eta \circ k, s) = [\![t_0]\!] \circ \eta \circ \mathsf{ff}$, Therefore,

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{acc}((r')^*, \mathsf{kkstr}(k,s), s)} \\
= & [\![3t_{\text{app}} + t_{\text{rcase}}]\!] \circ & \mathbf{cond}(\eta \circ \mathsf{ff})\ \mathbf{of} \\
& & \quad \mathbf{True:} \quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
& & \quad \mathbf{False:} \quad [\![2t_{\text{app}} + t_{\text{rec}} + t_{\text{false}}]\!] \circ \mathsf{acc}(r', \mathsf{checkk}(k,s,r'), s) \\
= & \multicolumn{2}{l}{[\![5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}} + t_{\text{false}}]\!] \circ [\![t + \sum_{j=1}^{n} t'_j]\!] \circ \eta \circ \mathsf{ff}} \\
= & \multicolumn{2}{l}{[\![5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}} + t_{\text{false}} + t + \sum_{j=1}^{n} t'_j]\!] \circ \eta \circ \mathsf{ff}}
\end{array}
$$

so, if $t'' = 5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}} + t_{\text{false}} + t + \sum_{j=1}^{n} t'_j$,

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{acc}((r')^*, k, s)} \\
= & [\![3t_{\text{app}} + t_{\text{rcase}}]\!] \circ & \mathbf{cond}([\![t_0]\!] \circ \eta \circ \mathsf{ff})\ \mathbf{of} \\
& & \quad \mathbf{True:} \quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
& & \quad \mathbf{False:} \quad [\![2t_{\text{app}} + t_{\text{rec}} + t_{\text{false}}]\!] \circ \mathsf{acc}(r', \mathsf{checkk}(k,s,r'), s) \\
= & \multicolumn{2}{l}{[\![5t_{\text{app}} + t_{\text{rcase}} + t_{\text{rec}} + t_{\text{false}} + t + \sum_{j=1}^{n} t'_j]\!] \circ [\![t_0]\!] \circ \eta \circ \mathsf{ff}} \\
= & \multicolumn{2}{l}{[\![t'']\!] \circ \mathbf{apply}(\eta \circ k, s)}
\end{array}
$$

- $i = 1$ and k is not false on s. Then either $\mathbf{apply}(\eta \circ k, s) = \perp$ or, there exists a cost t_0 such that $\mathbf{apply}(\eta \circ k, s) = [\![t_0]\!] \circ \eta \circ \mathsf{tt}$. Either way, $\mathbf{apply}(\mathsf{kkstr}(k,s), s) = \eta \circ \mathsf{tt}$, so

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{acc}((r')^*, \mathsf{kkstr}(k,s), s)} \\
= & [\![3t_{\text{app}} + t_{\text{rcase}}]\!] \circ & \mathbf{cond}(\eta \circ \mathsf{tt})\ \mathbf{of} \\
& & \quad \mathbf{True:} \quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
& & \quad \mathbf{False:} \quad [\![2t_{\text{app}} + t_{\text{rec}} + t_{\text{false}}]\!] \circ \mathsf{acc}(r', \mathsf{checkk}(k,s,r'), s) \\
= & \multicolumn{2}{l}{[\![t_{\text{true}} + 3t_{\text{app}} + t_{\text{rcase}}]\!] \circ \eta \circ \mathsf{tt}}
\end{array}
$$

Therefore, for $t = t_{\text{true}} + 2t_{\text{app}} + t_{\text{rcase}}$, if $\mathbf{apply}(\eta \circ k, s) = \perp$,

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{acc}((r')^*, \mathsf{kkstr}(k,s), s)} \\
= & [\![3t_{\text{app}} + t_{\text{rcase}}]\!] \circ & \mathbf{cond}(\perp)\ \mathbf{of} \\
& & \quad \mathbf{True:} \quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
& & \quad \mathbf{False:} \quad [\![2t_{\text{app}} + t_{\text{rec}} + t_{\text{false}}]\!] \circ \mathsf{acc}(r', \mathsf{checkk}(k,s,r'), s) \\
= & \perp & \\
= & \multicolumn{2}{l}{[\![t]\!] \circ \mathbf{apply}(\eta \circ k, s)}
\end{array}
$$

otherwise, if $\mathbf{apply}(\eta \circ k, s) = [\![t_0]\!] \circ \eta \circ \mathsf{tt}$, then

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{acc}((r')^*, \mathsf{kkstr}(k,s), s)} \\
= & [\![3t_{\text{app}} + t_{\text{rcase}}]\!] \circ & \mathbf{cond}([\![t_0]\!] \circ \eta \circ \mathsf{tt})\ \mathbf{of} \\
& & \quad \mathbf{True:} \quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
& & \quad \mathbf{False:} \quad [\![2t_{\text{app}} + t_{\text{rec}} + t_{\text{false}}]\!] \circ \mathsf{acc}(r', \mathsf{checkk}(k,s,r'), s) \\
= & \multicolumn{2}{l}{[\![t]\!] \circ [\![t_0]\!] \circ \eta \circ \mathsf{tt}} \\
= & \multicolumn{2}{l}{[\![t]\!] \circ \mathbf{apply}(\eta \circ k, s)}
\end{array}
$$

proving that the theorem holds in this case.

- $i > 1$. Then k must be false on s, so there exists a cost t_0 such that $\mathbf{apply}(\eta \circ k, s)$ equals $[\![t_0]\!] \circ \eta \circ \mathsf{ff}$.

  Let $[s''_1, \ldots, s''_{n'}]$ be the sublist of $[s_1, \ldots, s_n]$ such that each s''_i is not equal to s. Then, as with sequencing, $[s, s'_1, \ldots, s'_m]$ can be partitioned as follows:

$$s|\overbrace{s'_1 \ldots s'_{i_1}}^{S_{\mathrm{cont}}(r,s''_1)}|\overbrace{s'_{i_1+1} \ldots s'_{i_2}}^{S_{\mathrm{cont}}(r,s''_2)}|\ldots|\overbrace{s'_{i_{(n'-1)}+1} \ldots s'_m}^{S_{\mathrm{cont}}(r,s''_{n'})}$$

  Therefore there exists a j'' such that s'_{i-1} (the i'th entry) is in the partition $s'_{i_{j''-1}+1} \ldots s'_{i_{j''}}$. Now for all $i'' < j''$, k is false on $s'_{i_{i''-1}+1}$ through $s'_{i_{i''}}$, so by the induction hypothesis (as each $s''_{i''}$ is by assumption smaller than s),

$$\mathsf{acc}((r')^*, k, s''_{i''}) = [\![t''_{i''} + \sum_{h'=i_{(i''-1)}+1}^{i_{i''}-1} t'_{h'}]\!] \circ \mathbf{apply}(\eta \circ k, s'_{i_{i''}}) = [\![t''_{i''} + \sum_{h'=i_{(i''-1)}+1}^{i_{i''}} t'_{h'}]\!] \circ \eta \circ \mathsf{ff}$$

  where $\mathsf{acc}((r')^*, \mathsf{kkstr}(k, s'_{i_{i''}}), s''_{i''}) = [\![t''_{i'}]\!] \circ \eta \circ \mathsf{kkstrof}(k, s'_{i_{i''}}) = [\![t_{i''}]\!] \circ \eta \circ \mathsf{ff}$ and, for $i_{(i''-1)}+1 \leq h' \leq i_{i''}$, $\mathbf{apply}(k, s'_{h'}) = [\![t'_{h'}]\!] \circ \eta \circ \mathsf{ff}$. Because k is false on $s'_{i_{i''}}$, $\mathsf{kkstr}(k, s'_{i_{i''}})$ is false on all strings. Furthermore, by the definition of i, either k is false on s'_{i-1} or for all $i_{(i''-1)}+1 \leq h' \leq i_{i''}$, $s'_{h'}$ is not equal to s'_{i-1}. Therefore $\mathsf{kkstr}(k, s'_{i-1})$ is also false on $s'_{h'}$, so

$$\mathsf{acc}((r')^*, \mathsf{kkstr}(k, s'_{i-1}), s''_{i''}) = [\![t''_{i''}]\!] \circ \eta \circ \mathsf{ff}$$

  as well. For j'', we know that for $i_{(j''-1)}+1 \leq h' < i-1$, there exists a cost $t'_{h'}$ such that $\mathbf{apply}(\eta \circ k, s'_{h'}) = [\![t'_{h'}]\!] \circ \eta \circ \mathsf{ff}$, and that by the induction hypothesis

$$\mathsf{acc}((r')^*, k, s''_{j''}) = [\![t''_{j''} + \sum_{h'=i_{(j''-1)}}^{i-2} t'_{h'}]\!] \circ \mathbf{apply}(\eta \circ k, s'_{i-1})$$

  where $\mathsf{acc}((r')^*, \mathsf{kkstr}(k, s'_{i-1}), s''_{j''}) = [\![t''_{j''}]\!] \circ \eta \circ \mathsf{kkstrof}(k, s'_{i-1})$.

  To use this with r', we need to find the cost associated with applying both $\mathsf{checkk}(k, s, r')$ and $\mathsf{checkk}(\mathsf{kkstr}(k, s'_{i-1}), s, r')$ to strings in $S_{\mathrm{cont}}(r', s)$. For $i'' < j''$,

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{apply}(\eta \circ \mathsf{checkk}(\mathsf{kkstr}(k, s'_{i-1}), s, r'), s''_{i''})} \\
= & [\![2t_{\mathrm{snd}} + t_{\mathrm{app}}]\!] \\
 & \quad \circ\ \mathbf{cond}(\{\mathrm{false}\})\ \mathbf{of} \\
 & \qquad \mathbf{True:}\quad [\![t_{\mathrm{true}} + t_{=}]\!] \circ \eta \circ \mathsf{ff} \\
 & \qquad \mathbf{False:}\quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}} + t_{\neq}]\!] \circ \mathsf{acc}((r')^*, \mathsf{kkstr}(k, s'_{i-1}), s''_{i''}) \\
= & [\![3t_{\mathrm{app}} + 2t_{\mathrm{snd}} + t_{\mathrm{rec}} + t_{\mathrm{false}} + t_{\neq} + t''_{i''}]\!] \circ \eta \circ \mathsf{ff} \\
\multicolumn{2}{l}{\mathbf{apply}(\eta \circ \mathsf{checkk}(k, s, r'), s''_{i''})} \\
= & [\![3t_{\mathrm{app}} + 2t_{\mathrm{snd}} + t_{\mathrm{rec}} + t_{\mathrm{false}} + t_{\neq} + t''_{i''} + \sum_{h'=i_{i''-1}+1}^{i_{i''}} t'_{h'}]\!] \circ \eta \circ \mathsf{ff}
\end{array}$$

Also, for j'',

$$
\begin{array}{l}
\mathbf{apply}(\eta \circ \mathsf{checkk}(\mathsf{kkstr}(k, s'_{i-1}), s, r'), s''_{j''}) \\
\quad = [\![2t_{\text{snd}} + t_{\text{app}}]\!] \\
\qquad \circ\ \mathbf{cond}(\{\text{false}\})\ \mathbf{of} \\
\qquad\quad \mathbf{True:}\quad [\![t_{\text{true}} + t_{=}]\!] \circ \eta \circ \mathsf{ff} \\
\qquad\quad \mathbf{False:}\quad [\![2t_{\text{app}} + t_{\text{rec}} + t_{\text{false}} + t_{\neq}]\!] \circ \mathsf{acc}((r')^*, \mathsf{kkstr}(k, s'_{i-1}), s''_{j''}) \\
\quad = [\![3t_{\text{app}} + 2t_{\text{snd}} + t_{\text{rec}} + t_{\text{false}} + t_{\neq} + t''_{j''}]\!] \circ \eta \circ \mathsf{kkstrof}(k, s'_{j'}) \\
\mathbf{apply}(\eta \circ \mathsf{checkk}(k, s, r'), s''_{j''}) \\
\quad = [\![3t_{\text{app}} + 2t_{\text{snd}} + t_{\text{rec}} + t_{\text{false}} + t_{\neq} + t''_{j''} + \sum_{h'=i_{j''-1}+1}^{i-2} t'_{h'}]\!] \circ \mathbf{apply}(\eta \circ k, s'_{i-1})
\end{array}
$$

Furthermore, when the string does not change, we have that for any continuation k'

$$\mathbf{apply}(\eta \circ \mathsf{checkk}(k', s, r'), s) = [\![2t_{\text{snd}} + t_{\text{app}} + t_{\text{true}} + t_{=}]\!] \circ \eta \circ \mathsf{ff}$$

Let j refer to the element of $S_{\text{cont}}(r', s)$ corresponding to $s''_{j''}$, i.e., $s_j = s''_{j''}$. For $1 \leq h \leq j$, if $s_h = s$, then let $t_h = 2t_{\text{snd}} + t_{\text{app}} + t_{\text{true}} + t_{=}$, otherwise there exists an h'' such that $s_h = s''_{h''}$ and let $t_h = 3t_{\text{app}} + 2t_{\text{snd}} + t_{\text{rec}} + t_{\text{false}} + t_{\neq} + t''_{h''}$. Next note that by the equations above, for $1 \leq h < j$,

$$
\begin{array}{l}
\mathbf{apply}(\eta \circ \mathsf{kkstr}(\mathsf{checkk}(\mathsf{kkstr}(k, s'_{i-1}), s, r'), s_h)) \\
\quad = \mathbf{apply}(\eta \circ \mathsf{kkstr}(\mathsf{checkk}(k, s, r'), s_h)) \\
\quad = \eta \circ \mathsf{ff}
\end{array}
$$

and

$$
\begin{array}{l}
\mathbf{apply}(\eta \circ \mathsf{kkstr}(\mathsf{checkk}(\mathsf{kkstr}(k, s'_{i-1}), s, r'), s_j)) \\
\quad = \mathbf{apply}(\eta \circ \mathsf{kkstr}(\mathsf{checkk}(k, s, r'), s_j)) \\
\quad = \eta \circ \mathsf{kkstrof}(k, s'_{i-1})
\end{array}
$$

Thus, if k' is either $\mathsf{kkstr}(\mathsf{checkk}(\mathsf{kkstr}(k, s'_{i-1}), s, r'), s_h)$ or $\mathsf{kkstr}(\mathsf{checkk}(k, s, r'), s_h)$, there exists a single cost t' such that

$$\mathsf{acc}(r', k', s) = [\![t']\!] \circ \eta \circ \mathsf{kkstrof}(k, s'_{i-1})$$

and, by the induction hypothesis,

$$
\begin{array}{l}
\mathsf{acc}(r', \mathsf{checkk}(\mathsf{kkstr}(k, s'_{i-1}, s, r'), s)) \\
\quad = [\![t' + \sum_{h=1}^{j-1} t_h]\!] \circ \mathbf{apply}(\eta \circ \mathsf{checkk}(\mathsf{kkstr}(k, s'_{i-1}), s, r'), s_j) \\
\quad = [\![t' + \sum_{h=1}^{j} t_h]\!] \circ \eta \circ \mathsf{kkstrof}(k, s'_{i-1}) \\
\mathsf{acc}(r', \mathsf{checkk}(k, s, r'), s) \\
\quad = [\![t' + \sum_{h=1}^{j} t_h + \sum_{h'=1}^{(i-2)} t'_{h'}]\!] \circ \mathbf{apply}(\eta \circ k, s'_{i-1})
\end{array}
$$

Therefore

$$
\begin{array}{l}
\mathsf{acc}((r')^*, \mathsf{kkstr}(k, s'_{i-1}), s) \\
\quad = [\![t_{\text{app}}]\!] \circ\ \mathbf{cond}(\eta \circ \mathsf{ff})\ \mathbf{of} \\
\qquad\quad \mathbf{True:}\quad [\![t_{\text{true}}]\!] \circ \eta \circ \mathsf{tt} \\
\qquad\quad \mathbf{False:}\quad [\![2t_{\text{app}} + t_{\text{rec}} + t_{\text{false}}]\!] \circ \mathsf{acc}(r', \mathsf{checkk}(\mathsf{kkstr}(k, s'_{i-1}), s, r'), s) \\
\quad = [\![3t_{\text{app}} + t_{\text{rec}} + t_{\text{false}} + t' + \sum_{h=1}^{j} t_h]\!] \circ \eta\eta \circ \mathsf{kkstrof}(k, s'_{i-1})
\end{array}
$$

so let $t = 3t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}} + t' + \sum_{h=1}^{j} t_h$. Lastly, let $t'_0 = t_0$ (the cost of applying k to s). Then

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{acc}((r')^*, k, s)} \\
\quad = & [\![t_{\mathrm{app}}]\!] \circ & \mathbf{cond}([\![t'_0]\!] \circ \eta \circ \mathsf{ff})\ \mathbf{of} \\
 & & \quad \mathbf{True:} \quad [\![t_{\mathrm{true}}]\!] \circ \eta \circ \mathsf{tt} \\
 & & \quad \mathbf{False:} \quad [\![2t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}}]\!] \circ \mathsf{acc}(r', \mathsf{checkk}(k, s, r'), s) \\
\quad = & \multicolumn{2}{l}{[\![3t_{\mathrm{app}} + t_{\mathrm{rec}} + t_{\mathrm{false}} + t'_0 + t' + \sum_{h=1}^{j} t_h + \sum_{h'=1}^{(i-2)} t'_{h'}]\!] \circ \mathbf{apply}(\eta \circ k, s'_{i-1})} \\
\quad = & \multicolumn{2}{l}{[\![t + \sum_{h=0}^{i-2} t'_h]\!] \circ \mathbf{apply}(\eta \circ k, s'_{i-1})}
\end{array}
$$

thus showing that the theorem holds in this case.

□

Bibliography

[1] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF (extended abstract). In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software. International Symposium TACS'94*, number 789 in Lecture Notes in Computer Science, pages 1–15, Sendai, Japan, April 1994. Springer-Verlag.

[2] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[3] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall International, 1990.

[4] Michael Barr. Fixed points in cartesian closed categories. *Theoretical Computer Science*, 70:65–72, 1990.

[5] B. Bjerner. *Time Complexity of Programs in Type Theory*. Ph.D. thesis, Chalmers University of Technology, Sweden, 1989.

[6] Bror Bjerner and Sören Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*. Association for Computing Machinery, 1989.

[7] Geoffrey Burn and Daniel le Métayer. Proving the correctness of compiler optimisations based on a global analysis: a study of strictness analysis. *Journal of Functional Programming*, 6(1):75–109, January 1996.

[8] Andrzej Filinski. Representing monads. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 446–457. ACM, ACM, January 1994.

[9] Gustavo Gómez and Yanhong A. Liu. Automatic time-bound analysis for a higher-order language. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-02)*, volume 37, 3 of *ACM SIGPLAN Notices*, pages 75–86, New York, January 14–15 2002. ACM Press.

[10] John Greiner. *Semantics-based parallel cost models and their use in provably efficient implementations*. PhD thesis, Carnegie Mellon University, April 1997.

[11] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of computing. MIT Press, 1992.

[12] Douglas J. Gurr. *Semantic Frameworks for Complexity*. PhD thesis, University of Edinburgh, January 1991.

[13] John Hughes. Projections for polymorphic strictness analysis. In *Category Theory and Computer Science*. Springer-Verlag, 1989.

[14] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):97–107, 1989.

[15] H. Huwig and A. Poigne. A note on inconsistencies caused by fixpoints in a Cartesian closed category. *Theoretical Computer Science*, 73(1):101–112, June 1990.

[16] C. B. Jay, M. I. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. *Lecture Notes in Computer Science*, 1300:650–??, 1997.

[17] G. M. Kelly. *Basic Concepts of Enriched Category Theory.* Cambridge University Press, 1982.

[18] David King and Philip Wadler. Combining monads. In *Glasgow functional programming workshop*, July 92.

[19] J. Lambek. From λ-calculus to cartesian closed categories. In J.P. Selden and J.R. Hindley, editors, *To H. B. Curry : essays on combinatory logic, lambda calculus, and formalism*, pages 375–402. Academic Press, 1980.

[20] John Launchbury. A natural semantics for lazy evaluation. In *20th Annual ACM Symposium on Principles of Programming Languages*, pages 144–154. Association for Computing Machinery, January 1993.

[21] Daniel Le Métayer. Mechanical analysis of program complexity. In *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 69–73. Association for Computing Machinery, June 1985.

[22] Daniel Le Métayer. Ace: an automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10:248–266, April 1988.

[23] S. Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[24] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual IEEE Symposium on Logic in Computer Science (Asilomar, CA)*, pages 14–23. IEEE Computer Society Press, June 1989.

[25] E. Moggi. A modular approach to denotational semantics. In David H. Pitt, Pierre-Louis Curien, Samson Abramsky, Andrew M. Pitts, Axel Poigne, and David E. Rydeheard, editors, *Proceedings of Category Theory and Computer Science*, volume 530 of *LNCS*, pages 138–139, Berlin, Germany, September 1991. Springer.

[26] E. Moggi. Notions of computation and monads. *Information and Computation*, 1991.

[27] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 66–77, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[28] Philip S. Mulry. Categorical fixed point semantics. *Theoretical Computer Science*, 70(1):85–97, January 1990.

[29] Chris Okasaki. The role of lazy evaluation in amortized data structures. *ACM SIGPLAN Notices*, 31(6):62–72, June 1996.

[30] F. J. Oles. Type categories, functor categories and block structure. In M. Nivat amd J. C. Reynolds, editor, *Algebraic Semantics*, pages 543–574. Cambridge University Press, 1985.

[31] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

[32] A. M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[33] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation.* Cambridge University Press, 1995. Based on lectures given at the CLICS-II Summer School on Semantics and Logics of Computation, Isaac Newton Institute for Mathematical Sciences, Cambridge UK, September 1995.

[34] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland.

[35] John C. Reynolds. Using functor categories to generate intermediate code. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 25–36, San Francisco, California, January 22–25, 1995. ACM Press.

[36] M. Rosendahl. Automatic complexity analysis. In *Proc. of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. Association for Computing Machinery, September 1989.

[37] David Sands. *Calculi for Time Analysis of Functional Programs.* PhD thesis, University of London, September 1990.

[38] D. Scott. Domains for denotational semantics. In *Proceedings of ICALP 82*, volume 42 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, 1982.

[39] D. S. Scott. A type–theoretic alternative to CUCH, ISWIM, OWHY. Manuscript, 1969.

[40] Jill M. Seaman. *An Operational Semantics of Lazy Evaluation.* PhD thesis, Penssylvania State University, December 1993.

[41] Jon Shultis. On the complexity of higher-order programs. Technical Report CU-CS-288-85, University of Colorado, Boulder, 1985.

[42] I. Stark. Categorical models for local names. *J. Lisp and Symb. Comp.*, 9(1):77–107, February 1996.

[43] Robert D. Tennent. *Semantics of Programming Languages.* Prentice Hall, New York, 1991.

[44] P. Wadler. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, pages 385–407. Springer-Verlag, 1987.

[45] Philip Wadler. Strictness analysis aids time analysis. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 119–132. Association for Computing Machinery, January 1988.

[46] Philip Wadler. The essence of functional programming. In *Principles of Programming Languages*, Jan 1992.

[47] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI series, Series F: Computer and System Sciences*. Springer Verlag, 1994. Proceedings of the International Summer School at Marktoberdorf directed by F. L. Bauer, M. Broy, E. W. Dijkstra, D. Gries, and C. A. R. Hoare.

[48] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995. (This is a revised version of [47].).

[49] Mitchell Wand. Fixed-point constructions in order-enriched categories. *Theoretical Computer Science*, 8:13–30, 1979.

[50] B. Wegbreit. Mechanical program analysis. *Comm. of the ACM*, 18(9):528–539, September 1975.

[51] Stuart Charles Wray. *Implementation and Programming Techniques for Functional Languages*. PhD thesis, University of Cambridge, January 1986.