Cost-sensitive programming, verification, and semantics

Yue Niu

CMU-CS-24-153

September 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

This work was funded by the United States Air Force Office of Scientific Research under grants, MURI FA9550-21-0009, FA9550-21-1-0385 and the National Science Foundation under grant CCF-1901381. The author has also been supported by the Air Force Research Laboratory through the NDSEG fellowship. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of AFOSR, AFRL, or NSF.

Keywords: Compositional cost analysis, dependent type theory, program verification, denotational cost semantics, synthetic domain theory

Abstract

Although the pure *functional* semantics of computer programs has been well-studied since at least the seminal work of Scott and Strachey, it has remained challenging to integrate cost structure and the ability to speak about *cost-sensitive* properties of programs without compromising pure functional reasoning. This thesis contributes an approach to coherently integrating cost and functional verification in the setting of dependent type theory. Inspired by the method of *synthetic phase distinctions* of Sterling and Harper, I explain how the internal modal type theory of (pre)sheaf categories evinces a phase distinction between a cost-sensitive phase and a function phase suitable for such an integration. At the level of programming and verification, I demonstrate the ability of the internal type theory to mediate between cost-sensitive and purely functional verification. At the level of semantics, I prove an internal, cost-sensitive version of a classical result of Plotkin's — computational adequacy for **PCF**.

ACKNOWLEDGEMENTS

I want to thank Bob for inspiring me to study programming languages and spending six years guiding me to become a better researcher. Were it not for Bob's class(es) I probably would not have discovered or cared to learn about the theory of programming languages, a field that turned out to perfectly satisfy both my interest in mathematics and computer science. I often think back to the four-hour meetings in which we would discuss technical problems and feel grateful to have an advisor who is so invested in the development of his students.

Speaking of students of Bob, I must thank Jon. Over the last few years, I have learned a tremendous amount from Jon, who has also been a major source of inspiration to my mathematical style and scientific point of view. It would not be an overstatement that this thesis would not exist in its current form without the (direct and indirect) intellectual contributions from Jon. I also thank Carlo, from whose thesis I learned about the computational semantics of type theory.

My gratitude also goes to Jan for taking me on as an undergraduate student, showing me how to navigate the dense forest of programming languages literature, and teaching me the tools of the trade. I am also grateful to my colleagues and friends at CMU. I enjoyed many conversations on various PL-adjacent topics with Harrison, who also has a keen ability to clearly explain difficult ideas. I appreciate Long, Jatin, David, and Siva for their support and camaraderie over the years.

Outside of my academic life, I am indebted to my parents for supporting my decision to pursue a PhD, a career choice that has dubious financial prospects. For this I thank (blame?) Jona and Paul, who helped me to convince myself to do a PhD in programming languages one fateful night. Last but not least, I am grateful to my friends in 729 and my dearest confidant "Monchey" — may we all congregate again soon.

Contents

Prelude

(0∗1) This monograph is organized into "nodes" such as **(0∗1)**. Some of them are annotated with icons to help distinguish at a glance the nature of the node:

📖 definitions.
□ propositions, lemma, theorems.
■ proofs.
⏬ corollaries.
🔨 constructions.
✎ examples.
🏋 exercises.
👇 remarks that serve to build intuition.
💡 insights.
↗ references.
ℹ auxiliary facts.
⚠ caution advised.

0.1. COMPUTER PROGRAMS AND MATHEMATICAL SEMANTICS

(0.1∗1) Ostensibly this dissertation is about program verification and cost analysis. But what does this mean? Often there is a significant gap between the kind of problems considered by researchers of programming languages and the perception of the situation from outside the clique of PL researchers. The purpose of this first chapter is to give a sense of the context and scope of the theory of programming languages as it pertains to program verification and cost analysis.

(0.1∗2) When confronted with skepticism about the purported behavior of a program or procedure (often as a component of a larger piece of software), intuitively one might begin justification with "well, because this part of the program does this,...". At this point both the questioning and answer stand on shaky ground; until one unambiguously determines what it means for a program to "do", it is very easy to trick oneself into believing properties about programs that are easily falsifiable.

(0.1∗3) To take a simple example, consider the following program snippets in a context equipped with two memory cells `x,y` containing integer values:

```
x = y + y                          x = 2 * y
```

It seems reasonable to conclude that the value of the cell `x` remains unchanged in all contexts if we replace the first snippet with the second. But this is not true in a *concurrent* or multi-threaded program because the value of `y` is read twice in the first snippet while it is only read once in the second. A distinction between the effects of the two may be observed when the value of `y` is mutated between the two read actions in the first snippet.

(0.1∗4) The example in **(0.1∗3)** illustrates that even the simplest programs can introduce "bugs", in other words unintended program behaviors. In this particular case, the source of the problem is the confusion between program *variables* that may be manipulated using familiar algebraic equations and *names of memory cells*, which may not. However, interactions amongst memory cells can also be manipulated algebraically using the equations of a *different* algebraic theory.

(0.1∗5) Thus a primary source of questions in programming languages is furnished by the search for natural semantics of programs in terms of existing mathematical structures or the inventions of new structures (in the case of the theory of memory cells). The informal description of what a program "does" is rationalized as the *meaning* or *denotation* of a program in a mathematical model. The benefit of this arrangement is that questions about program behavior may be phrased as mathematical propositions for which one has a definitive method for determining what counts as proof. Therefore we may reduce the problem of giving a rigorous specification of programs and verifying such a specification to that of mathematical proof. This reduction not only puts the study of programming languages on firm logical footing but also brings to bear extant mathematical theories and techniques on specific PL problems of interest.

(0.1∗6) The primordial and most important mathematical model of a program is the *function.* If we know that a program `e` is assigned a function $f : A \to B$ as its meaning[1], then we know that `e` "computes" f in the following sense: for all inputs `x` whose meaning is $a \in A$, the meaning of the composite program e supplied with input `x` is $f(a) \in B$. One can think of the domain and codomain of f as representing the sets of allowable program behaviors and the meaning of a program as a mapping between allowable behaviors.

To illustrate by way of example, let us consider one of the earliest known programs, *Euclid's algorithm* `euclid` for computing the greatest common divisor; `euclid` denotes the function $gcd : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$:

$$gcd(n, m) = \begin{cases} n & \text{if } m = 0 \\ gcd(m, n \bmod m) & \text{o.w.} \end{cases} \tag{0.1∗6∗1}$$

We may call the declaration $gcd : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ a *function signature*, which when viewed as the meaning of a program describes a behavioral specification of the program. In this case, it tells us that given an input whose meaning is a natural number, `euclid` applied to that input is a program whose denotation is a natural number as well. Not only this, we also know that if `n,m` represents natural numbers n, m, then the instance of `euclid` on the inputs `n,m` will denote the natural number $gcd(n, m)$. Thus one may reasonably say that `euclid` *implements* the *function* that sends two numbers to their greatest common divisor.

(0.1∗7) Functions are useful as a concept because they may be composed: functions $f : A \to B$ and $g : B \to C$ with matching codomain and domain compose to a function $f; g : A \to C$ that effects the constituent functions in sequence. It is difficult to overstate the utility of function composition in programming. It enables program components to be developed modularly and

[1]We defer the question of *how* one can come to know about such an assignment to a later time.

independently, which are essential characteristics of the development of large-scale software requiring the cooperation of people. The overall desired behavior of a system may be carefully broken down into smaller function signatures that can be implemented in isolation from each other and eventually composed along the specified boundaries. The notion of functions not only fulfills the demand for modularity in practice but also reduces the "highwater-mark" of the mental load required of a single programmer.

0.2. NATURE OF PROGRAMS

(0.2∗1) So far I have said suspiciously little about what programs actually *are* and instead relied on the reader's experience and intuition to contextualize what phrases such as "supplied with input" and "instance" must mean when referring to computer programs. This is not an accident. Considering the development of the subject, it is quite expected that generally speaking, it is easier to explain what a program *means* than to explain what a program *is*[2]. We shall try to arrive at what a program ought to be, under the current circumstances.

(0.2∗2) A clue comes from Eq. (0.1∗6∗1). Although reasonable at first glance, the definition of *gcd* appears suspect upon further inspection: how come one is allowed to invoke the definition of the very function one is in the process of defining? For instance, we may consider a similar use of self-reference in the following:

$$f(x) = f(x+1) \tag{0.2∗2∗1}$$

What is f? In particular, we observe that the "definition" above does not uniquely characterize a function $\mathbb{N} \to \mathbb{N}$. For instance, any constant function g will have the property that $g(x) = (x+1)$. Mathematically we may say f is under-specified or ill-defined, which calls alarm to the legitimacy of self-reference as a tool for defining functions.

(0.2∗3) What makes Eq. (0.1∗6∗1) a well-defined function and Eq. (0.2∗2∗1) ill-defined? Calling for the moment a candidate to a function an "expression", one can show by mathematical induction that the former expression defines a valid function; no such method is available for the latter. A *programming language* is a collection of rules that systematizes such justifications of validity. A *program* is an expression that denotes a function. The purpose of rules is to *make routine*[3] the problem of verifying when an expression defines a function, which paves the way for thinking about problems at a higher level of abstraction.

(0.2∗4) When defining a programming language, one not only delineates programs by stipulating which expressions are valid but also specifies when two programs are equal, *i.e.* denote the same function. The equational theory of programs furnishes an abstract notion of computation through which one can exhibit deductions like $2+2=4$. (Remember, $2+2$ and 4 are *programs* that are written in a semantically-inspired notation. I could have written "$\mathit{green}(\mathit{apple}, \mathit{apple}) = \mathit{orange}$" and you would probably question why that should be the case. The point is that the equational theory should be true in the intended mathematical model.)

[2] I will concede that a computer programmer who has had no conception of the notion of a function (however unlikely) may well believe the opposite.

[3] One can say that a programming language is a syntactic discipline for enforcing semantic invariants; in terms of implementations of programming languages, one can think about these rules as the rules followed by a "type checker".

0.3. BREAKDOWN OF THE FUNCTIONAL SEMANTICS

(0.3∗1) Insofar as we believe in the model of programs as denoting functions, program specification and verification are no different from proving any other mathematical theorem. But this model also validates properties of programs that seem bewildering when we take into account the idiosyncratic tendency for programs to consume resources when executed, an observation that serves as the foundation of the field of computational complexity and cost analysis.

(0.3∗2) The actual resource consumption of programs is too noisy and complicated; the basic approach of cost analysis is to tie a predetermined, constant unit of resource to the execution of certain program components and to gain an approximate bound on the overall resource usage (such as time or memory) by tracking the movement of the basic instances of resource usage. A typical example would be to attach one unit of abstract resource to the comparison operation in sorting procedures. Under this cost model, one may prove a bound for insertion sort that is quadratic in the length of the input n and a $k \cdot n \log(n)$ bound for mergesort, where k is some constant independent of n. The justification and utility of these bounds is that the number of comparisons is a good predictor of the asymptotic runtime of the sorting procedure.

(0.3∗3) Taking the conjunction of **(0.1∗6)** and **(0.3∗2)**, we quickly arrive at the point of tension: if both insertion sort and mergesort are to denote some function $f : \mathbb{N}^* \to \mathbb{N}^*$ sending a list of numbers $n_1, \ldots n_k$ to $m_1, \ldots, m_k$ such that the m_i's are in order and forms a permutation of the n_i's, then they must denote the same function because the requirements above uniquely determine f. But this means that we have no way of distinguishing between insertion sort and mergesort in the model!

(0.3∗4) There are at least two approaches one may take. One can forgo the interpretation of programs as functions and instead exhibit dynamic behavior by means of relations of the form `e` $\mapsto$ `e'`, which represents the passage of an atomic step or the exhaustion of a basic unit of resource. In this model, we may define the cost of a program to be the maximal number of atomic steps. The downside is that ordinary program equivalences must be tracked by extra data witnessing the relation $\mathsf{e} \mapsto \mathsf{e}'$; moreover, atomic steps are only defined for *closed* programs (an expression with no free variables), and it is quite involved to make the theory of atomic steps to work with programs with inputs. Although this operational model of programs is not so useful from our current perspective, we will have more to say about its role in the overall vision of the work and relation to the functional model in Section 0.5.

(0.3∗5) A different approach would be to adjust the functional semantics of **(0.1∗6)** so that the meaning of a program is a function of the form $f : A \to \mathbb{C} \times B$, or equivalently a pair of functions $f_{\mathsf{val}} : A \to B$ and $f_{\mathsf{cost}} : A \to \mathbb{C}$. The idea is that f_{val} is the original functional semantics and f_{cost} is a function that sends an input to the cost of executing the program on that input. This model can correctly distinguish between the resource usage of insertion sort and mergesort, but the equational theory is now too stringent — sometimes we *want* to be able to equate insertion sort and mergesort on the basis that their f_{val} components are equal! This desire is not only useful for validating the pure functional correctness of programs but also critical for establishing cost bounds.

(0.3∗6) A main contribution of this dissertation is to rectify the semantics outlined in **(0.3∗5)**

so that one can *selectively* and *safely* ignore the f_{cost} component of the interpretation. We will describe this in more detail in Section 3.3.

0.4. HIGHER-ORDER COMPUTATION AND RECURSION

(0.4∗1) The modern understanding of computation emerged in the 1930s through a machine model known as Turing machines. Very roughly, a TM is a mathematical structure consisting of a finite set of *states* and a finite collection of *rules* that interacts with an infinite array of *binary memory cells*[4] equipped with an index called the *head position*. A rule takes as input the current state of the machine, the head position, and the content of the cell at that position and outputs an updated value to the cell along with a new state and head position. A *computation* is initiated by placing the input as a binary string at the beginning of the memory array and setting the head position to the first cell. The computation evolves according to the collection of rules, starting with a special rule governing a distinguished initial state and terminates when a distinguished stop state is reached. The output of the computation is whatever is left on the memory array.

An intuitive selling point of Turing machines as a model of computation is that an extremely dedicated and tireless person can carry out the rules of the machine, at least in principle. Perhaps due to this intuitive mechanical appeal, despite postceding several other notions of a computable function (miraculously, all of these turned out to be equivalent[5], including Turing machines), Turing machines persisted and became the now widely accepted foundation for the subject.

(0.4∗2) The classically trained computer scientist may question whether Turing machines furnish a reasonable notion of programs. Unfortunately, Turing machines are the wrong level of abstraction for the purpose of programming and verification. Setting aside common data structures such as lists and trees, even the notion of a number is clunkily encoded as a bit string when working with Turing machines. It is simply hopeless to try to formalize the complexities and requirements of modern software in terms of a machine model of computation.

To take as an example (which is also the real point of the current section), we can look at the pathological treatment of *higher-order functions* in the setting of Turing machines. A higher-order function or a *functional* is a function of functions, *e.g.* a function whose signature is $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$. In this context, one can cope with such higher-order data by *encoding* the input function as a Turing machine, which can be further encoded as a bit string that can be operated on as usual. But such an arrangement can hardly be said to compute a functional because the abstraction provided by the framework of Turing machines does not strictly enforce the *interface* of a function, namely, that of applying one with an input. To wit, one can define an exotic "higher-order" Turing machine whose output can differ between two encoded Turing machines $\mathcal{M}_1$ and $\mathcal{M}_2$ even when they encode the same $\mathbb{N} \to \mathbb{N}$ function. Therefore Turing machines support *higher-order machines* (that is, a Turing machine whose input is supposed to be the bit encoding of another Turing machine) rather than higher-order functions. Although the notion of Turing machines is inappropriate as a (user-facing) programming language, its influence becomes relevant again when we consider the *physical semantics* of programs in Section 0.5.

[4]It might be more accurate to say that the cells are ternary because blank cells are semantically significant.

[5]At least in the case of *first-order functions*; the notions diverge for higher-order computations, as we will soon see.

(0.4∗3) A main contribution of this dissertation is a compositional cost semantics of **PCF**, a programming language for *higher-order recursive functions.* In particular, we review the classic mathematical semantics of recursion and show how it may be extended in a cost-sensitive manner that is compatible with **(0.3∗5)**.

0.5. TOWARDS A PHYSICAL SEMANTICS OF PROGRAMS

(0.5∗1) The conception of programs and their semantics in Section 0.1 may be attractive and motivated from a mathematical perspective, but one may develop a gnawing feeling that this isn't what programs *actually do.* After all, real computers interpret analog electrical signals as binary bits and operate on them using a fixed set of arithmetic circuits. They certainly do not execute a program by computing its meaning in the sense of Section 0.1. How can we be sure that our idealized mathematical semantics predicts anything about the physical reality of computing?

(0.5∗2) One way to bridge the conceptual distance between the mathematical semantics and physical semantics of programs is to trace the various transformations a program undertakes when it is executed on a computer. Taking an extremely simplified view, there are three distinct phases in this transformation: 1) programs as "source code", which is what I have been developing so far; 2) programs as "target code" or the result of *compilation*, a process that translates source code into programs of another programming language; 3) programs as physical processes and signals that are interpreted and enacted upon according to the predetermined function of the hardware.

Logically these transformations can be modeled by stratifying both the programming language and its semantics into multiple layers reflecting the different levels of abstraction relevant to each stage of the transformation. For instance, we might refine the functional semantics in which we only consider function signatures of the form $f : \{0,1\}^* \to \{0,1\}^*$, *i.e.* mappings between finite bit sequences. The idea is that semantic objects in the functional semantics can be *encoded* as finite binary sequences, which might correspond to arrays of physical memory addressable by the hardware. Morally this is similar to the Turing machine model of computation discussed in **(0.4∗1)**.

(0.5∗3) The theory of programming languages is somewhat agnostic to the material nature of the stages of this transformation; I picked three representative stages in **(0.5∗2)** to give a "feel" in terms of familiar structures, but in principle there are no limitations to the kind of language and semantics one could consider. What is important is to connect the semantics between successive stages by means of an *adequacy* theorem. Roughly speaking adequacy is the agreement of different models of computations with respect to *observable* properties; the justification for this relaxation is that differences between models can only be detected in the real world by means of observations. Usually adequacy is proved relative to the canonical observation — termination.

(0.5∗4) In Chapter 8, I unfold the first "stage" of the kind of transformation discussed in **(0.5∗2)** for the programming language **PCF (0.4∗3)** by introducing a trivial kind of compilation process that leaves the program text unchanged but changes the interpretation from the functional kind described in **(0.1∗6)** to the one operational one described in **(0.3∗4)**. The main result of Chapter 8 is the proof of an adequacy property in the sense of **(0.5∗3)** that is *cost-sensitive*, which means that not only do the semantics agree on termination, they agree on the amount of resources incurred during computation.

The utility of such a theorem is to connect the functional semantical approach taken in this dissertation to extant research on cost analysis and verification, a large part of which is understood in operational terms. Although the naive operational models considered here are still highly unrealistic as models of real-world computers, this connection nonetheless brings us a step closer to validating the cost specifications of computer programs *qua* physical processes by means of high-level mathematical models.

Part I

Programming and verification

CHAPTER 1

Introduction

(1∗1) This thesis is an investigation into the semantics and verification of both *functional* and *cost-sensitive* properties of computer programs. A functional or correctness property is a specification about *what* a program is supposed to do, while a cost-sensitive property is a specification about *how much* computational resource a program requires to complete a task. Concretely the specification and verification of cost-sensitive properties can vary in several (mostly) orthogonal axes: the computational resource in question, the resource metric, and the notion of the cost bound.

A *computational resource* could refer to concrete measures like time or space usage, but could also refer to abstract measures like the number of critical operations incurred during program execution. A *resource metric* or *figure of merit* is the determination of what counts as incurring cost in a particular analysis. For example in the analysis of sorting algorithms it is common to take the number of comparisons used as the figure of merit. Such resource metrics may be called *nonuniform* or *algorithm-specific* because the metric only applies to the program under analysis. In contrast, one may take the figure of merit to be some general construct of the programming language (the number of β-reductions is often used in this scenario); such resource metrics may be called *uniform* to reflect the fact the metric applies uniformly at the language level.

Although the resource metric is often determined by the computational resource, technically speaking there is no reason that any computational resource must be accompanied by a particular metric or vice versa. For instance, when analyzing a sorting algorithm, wall time and the number of comparisons can be both used as *proxies* of the computational resource of time. Thus we can view a computational resource as an *informal* resource metric.

Lastly, one's approach to cost analysis can differ in the amount of precision required on the form of cost bounds: they can either be concrete (the cost can be bounded by a *specific* function of the inputs to the program) or asymptotic (there *exists* some unspecified bounding function in a class of functions). Observe that a concrete cost bound can always be immediately abstracted into an asymptotic bound but the converse requires additional work involving a careful quantitative refinement of the original proof of the asymptotic bound and is not necessarily possible depending on the way the original bound is derived.

(1∗2) In this thesis I develop a program verification framework for functional programs supporting all the variations on cost analysis discussed in **(1∗1)**. In particular I focus on the derivation of concrete cost bounds because 1) a concrete bound is (nearly) an asymptotic bound 2) the sort of simplifications afforded by asymptotic analysis of algorithms on paper do not translate well into a setting in which concrete bounds are often necessary (as is in the setting of formalized cost analysis).

(1∗3) As far as pure *functional* semantics and verification is concerned, both the computer science and mathematics community have been converging towards the use of *type-theoretic proof assistants* [26, 86, 91] to formally verify the properties of computer programs [23, 70, 128, 124] on the one hand and mathematical theorems [20, 48, 40] on the other.

(1∗4) Type theory is not a monolith. However one unifying perspective on type theories is put pithily in Reynolds [101].

> Type structure is a syntactic discipline for maintaining levels of abstraction.

One way to interpret Reynold's definition is that type theory is a mathematical language for describing a semantic domain such that the evidence for the validity of a judgment or property of an entity in the language is easily discernible. In this way type theories furnish a *syntactic* discipline for constructing correct observations about a potentially complicated semantic domain — an *abstraction*. The primary mechanism in type theories supporting this syntactic discipline is *composition*, which allows one to break down judgments and propositions into their constituent parts, each amenable to independent verification.

(1∗5) As a codification of both mathematical structures and computer programs, type theory has been a wildly successful vehicle for specifying and verifying functional/correctness properties. On the computer science side, proof assistants based on type theory have been used to verify critical components in the modern software stack including compilers [71], operating systems kernels [133], web servers [68], and concurrent-imperative programs [65]. On the mathematical side, type-theoretic proof assistants have also been used to push the boundary of mechanized mathematics, as demonstrated in the formalization of the four color theorem [40] and the liquid Tensor project [104].

(1∗6) The story for *cost-sensitive* properties is considerably less developed. In fact, the notion of computational cost is explicitly *excluded* in the CompCert project on verified compilers (emphasis added) [71].

> "What are observable behaviors?...They include everything the user of the program[...]can "see" about the actions of the program, with the notable exception of *execution time* and *memory consumption*."

Moreover, extant frameworks for defining and analyzing the cost structure of programs, type-theoretic or otherwise, suffer from various semantic deficiencies that compromise either the faithfulness of the representation of cost structure or ergonomics, *i.e.* the practical usability of the framework. In particular, most of the standard approaches to formalized cost analysis take one of three routes:

1. *Operational cost semantics* [7, 83]. In the operational setting programs are equipped with a system of transition relations $\mapsto_c$ annotated with some resource c. Usually one develops a program logic around the operational cost semantics to facilitate verification.

2. *Denotational cost semantics* [30, 49]. Alternatively one can model cost structure as an auxiliary output/input of programs, leading to the *profiling* semantics of cost structure. In this scenario it is common to think about cost as an abstract *computational effect* in the sense of Moggi [85], but it is also possible to represent cost structure concretely as in Handley, Vazou, and Hutton [49].

3. *Type systems for cost analysis* [57, 132, 97]. A category more distant to the approach taken in this thesis is that of specially designed type systems for *syntactically* deriving cost bounds of a certain class (usually polynomial), often in a completely automated manner.

(1*7) Each of the three perspectives outlined in **(1*5)** evinces problems that are not isolated to a particular framework but characteristic of the approach in general. In the operational setting it is typical to view programs as purely syntactic objects, which means one must reason about programs *indirectly* in terms of a conceptually intuitive but ultimately arbitrary system of rules about execution behavior. Despite the low technical barrier to providing a rigorous semantics for a variety of programming languages equipped with sophisticated static and dynamic features, operational semantics obfuscates what programs *are* — functions. An emblematic problem is that because operational (cost) semantics is only defined over closed terms, it takes considerable effort to extend the global equational theory of complete programs to a local theory on open terms.

(1*8) On the other hand, denotational cost semantics directly refines the functional semantics of programs by making cost structure (either concretely or abstractly) part of the input/output behavior. However integrating cost structure naively in terms of profiling/instrumentation can undermine the intended semantics of programs. For instance, we may define the following program *badList*, whose input and output are extended with a parameter $\mathbb{N}$ that represents resource usage.

$$\begin{aligned}&\mathit{badList} : \mathbb{N} \times \mathsf{list}(\mathbb{N}) \to \mathbb{N} \times \mathsf{list}(\mathbb{N})\\ &\mathit{badList}(c, l) = (c, c :: l)\end{aligned}$$

Observe that because the input *cost* parameter c is used to compute the output *functional behavior* $c :: l$, the program *badList* cannot have a coherent functional semantics independent of the cost profiling. Furthermore, because nothing about the specification $\mathbb{N} \times \mathsf{list}(\mathbb{N}) \to \mathbb{N} \times \mathsf{list}(\mathbb{N})$ distinguishes cost parameters from actual inputs, basic properties about the denotational cost semantics (such as increasing the amount of starting resource does not change the output) may be violated.

The situation improves when the profiling/instrumentation is treated abstractly in terms of a computational effect — for instance we may postulate a new type $\mathsf{F}A$ equipped with an operation $\mathsf{step} : \mathsf{F}A \to \mathsf{F}A$ that records the occurrence of a cost effect. Under the hood we may implement the instrumentation by defining $\mathsf{F}A = \mathbb{N} \times A$ and $\mathsf{step}(c, a) = (c + 1, a)$, but this is *not* revealed to the programmer. In other words, we think about cost structure as an *abstract interface*, which allows us to prove stronger safety properties since the programmer is disallowed from defining pathological programs such as *badList*.

However, in either the concrete or abstract approach to cost profiling one still faces the challenge of *stripping away* the profiling to obtain the purely functional semantics of programs. The ability to reason about programs up to purely functional behavior is not only inherently desirable but also critical to proving *cost-sensitive* properties; for instance, often times a cost bound will depend on some data structure invariant such as a tree being balanced, which is a purely functional property.

(1∗9) Lastly there is the approach to cost analysis by means of a type system that is specially designed to derive cost bounds by purely syntactic considerations. Type systems are compositional by design and rule out badly behaved programs such as *badList*, but they cannot serve as *general-purpose* verification frameworks since the equational theory of programs is often neglected in return for greater degrees of automation.

(1∗10) In this dissertation I contribute a type theory dubbed **c**ost-**a**ware **l**ogical **f**ramework (**calf**) for both defining and verifying functional and cost-sensitive properties that addresses the deficiencies of the extant approaches to cost-sensitive verification outlined in **(1∗7)** through **(1∗9)**. I aim to support the following thesis.

> The internal modal type theory of (pre)sheaf topoi furnishes an ergonomic language for both ***functional*** and ***cost-sensitive*** programming, verification, and semantics.

1.1. PROVENANCE

(1.1∗1) The content of this dissertation is derived from the following published works.

1. Yue Niu et al. “A Cost-Aware Logical Framework”. In: *Proceedings of the ACM on Programming Languages* 6.POPL (Jan. 2022). DOI: `10.1145/3498670`. arXiv: `2107.04663 [cs.PL]`.
2. Yue Niu and Robert Harper. “A Metalanguage for Cost-Aware Denotational Semantics”. In: *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2023, pp. 1–14. DOI: `10.1109/LICS56636.2023.10175777`.
3. Yue Niu, Jonathan Sterling, and Robert Harper. *Cost-sensitive computational adequacy of higher-order recursion in synthetic domain theory.* 40th Conference on Mathematical Foundations of Programming Semantics (MFPS XXXX). 2024. arXiv: `2404.00212 [cs.PL]`. URL: `https://arxiv.org/abs/2404.00212`.

1.2. OVERVIEW

1.2.1. Part I: programming and verification.

(1.2.1∗1) In the first part of the thesis I discuss and explain the technical tools used in this thesis. Chapters 2 and 3 consist of a review of existing work, while Chapter 4 is original work derived from the works cited in **(1.1∗1)**. In Chapter 2, I review the basics of dependent type theory, the syntactic discipline used to integrate cost-sensitive and functional program verification in this thesis. In Section 2.1 I recall the rules of this syntactic discipline and in Section 2.6 I outline a model construction of dependent type theory in sets.

(1.2.1∗2) Chapter 3 provides the necessary mathematical background used in later chapters, namely that of *categories of (pre)sheaves over a small base category*, which are the main source of semantic models of type theories used in this thesis. In Section 3.2 I explain how every presheaf category supports an *internal type theory*, simultaneously furnishing a functional programming language and an expressive, constructive logic that is suitable for both developing general mathematics and program specifications and verification. In Section 3.3 this internal language is enriched with the notion of a *phase distinction* in certain categories of presheaves that delineates the cost-sensitive and purely functional aspect of the associated internal type theory. Sections 3.4 and 3.5 contain material of a more technical flavor that becomes essential in the second half of the thesis on the (cost) *semantics* of programming languages.

(1.2.1*3) The preceding chapters on dependent type theory and the phase distinction culminate in a type theory for integrating cost-sensitive and functional program verification in Chapter 4, dubbed **calf** for a cost-aware logical framework. I outline the syntactic aspect of this theory in Section 4.2 and demonstrate how **calf** is used in practice in Section 4.3. In Section 4.4, I justify the logical consistency of **calf** by constructing a model in any category of presheaves equipped with a phase distinction.

1.2.2. Part II: semantics.

(1.2.2*1) The second part of this thesis concerns the *internal denotational cost semantics* of general recursion in type theory. In addition to possessing intrinsic mathematical value, internal denotational cost semantics provides a way to connect the denotational and operational approaches to cost analysis discussed in **(1*7)** and **(1*8)**. In contrast to prior work on denotational cost semantics in traditional categories of domains [66, 95], I develop the denotational cost semantics of general recursion inside *synthetic domain theory*, an approach to domain theory that seamlessly integrates into dependent type theory. The main result of this development is the axiomatization and model construction of a synthetic domain theory equipped with a phase distinction and the proof of an internal, cost-sensitive version of Plotkin's seminal *computational adequacy* property for **PCF** [96].

(1.2.2*2) In Chapter 6, I review classic domain theory and highlight some constructive and topological considerations relevant to synthetic domain theory. In Chapter 7 the notion of *axiomatic domain theory* is broached to pave the way toward synthetic domain theory; the general structure of the situation is explicated in Section 7.1 in terms of the simpler theory of *preorders*. In Section 7.3, I study the order-theoretic and closure properties of (pre)domains in synthetic domain theory. In Section 7.4, I validate the axioms introduced by means of a sheaf-theoretic model construction.

(1.2.2*3) The synthetic domain theory developed in Chapter 7 is deployed in Chapter 8 to define the internal denotational cost semantics of $\mathbf{PCF}_{\mathsf{cost}}$, a version of **PCF** equipped with an abstract cost effect. I prove that $\mathbf{PCF}_{\mathsf{cost}}$ satisfies a cost-sensitive computational adequacy theorem relative to a dynamic semantics of $\mathbf{PCF}_{\mathsf{cost}}$ dubbed the *computational semantics*; the relationship between this new computational semantics and traditional operational cost semantics is discussed in Section 8.5.

(1.2.2*4) In Chapter 9, I end with a discussion of related work and speculations of future work.

CHAPTER 2

Dependent type theory

(2∗1) The technical contributions of this dissertation are rooted in a particular scientific discipline involving the study of *dependent type theory*, usually shortened to just type theory. The purpose of this chapter is to give the reader an idea of the features of type theory and a feel for what is it like to use type theory. For a more principled approach we refer the reader to Martin-Löf [81] for the origins of type theory (in the modern sense) and Univalent Foundations Program [129] for a textbook introduction.

(2∗2) From a certain technical point of view, type theory is about the mathematics of *indexing*, or more precisely, it is the formal language for describing and manipulating structures that vary over some context. For instance, consider the following definition of a group:

> A *group* is a set G equipped with a distinguished element $e \in G$ and functions $\cdot : G \times G \to G$ and $-^{-1} : G \to G$ such that ...

Where in the above I have elided the well-known group axioms. Thus the structure of a group is captured by the *family* $(G \times (G \times G \to G) \times (G \to G) \times \dots)_{G \in \mathcal{U}}$ whose index of variation $\mathcal{U}$ is a given universe of discourse for the carrier sets. Dependency refers to the fact that the "shape" or *type* of the group operations *depends* on the carrier set. Dependent type theory allows one to rigorously specify and work with structures of this kind.

(2∗3) This *technical reason* motivating the notion of dependent types hides a wealth of deep and rich mathematics and obfuscates much of the history of the subject, but this monograph is not the right place to explain them. Instead, a more impressionistic account of type theory will be given, which reflects the author's personal opinion and scientific requirements. In the following, I will outline three perspectives or attitudes one might possess towards type theory: *algebraic*, *computational*, and *synthetic.*

2.1. THE ALGEBRAIC PERSPECTIVE

(2.1∗1) Just as there is more than one group, there is more than one type theory. Thus the question "What is a type theory?" is not answered by pointing to a particular instance but by giving a general characterization of the structure akin to the group axioms. The proposed responses typically involve mathematical structures of algebraic character; examples include contextual categories [21], categories with families [32], comprehension categories [63], locally Cartesian closed categories [108], *etc.* At a high level these models aim to capture the essential structures that must exist to make sense of the following *judgments* of type theory:

$\Gamma\ \mathsf{ctx}$
$\Gamma \vdash A\ \mathsf{type}$
$\Gamma \vdash M : A$
$\Gamma \vdash A = A'$
$\Gamma \vdash M = M' : A$

Roughly one can think of type judgments as nodes in an exceedingly complex (but well-founded) tree called a *derivation*; the branching of the tree is controlled by the *inference rules* of the type theory, which controls the branching of the derivation. Concretely, an inference rule takes the following form:

$$\frac{\mathcal{J}_1 \qquad \dots \qquad \mathcal{J}_n}{\mathcal{J}}$$

where $\mathcal{J}$ and every $\mathcal{J}_i$ is one of the judgments outlined above. The judgments $\mathcal{J}_i$ are called *premises* and $\mathcal{J}$ is called the conclusion of the inference. Finite applications of inference rules generate a derivation tree by the following procedure: take as root the *conclusion* of some inference; the premises of that rule are added as children, and repeat the procedure by taking some children as the conclusion of another inference. A derivation is *closed* when its leaves are labeled by judgments that are conclusions of axioms, *e.g.* inference rules with no premises. A judgment is *derivable* when it is the root of a closed derivation. For the purpose of discussion, we will call a type theory defined in terms of such inference rules an *algebraic type theory.*

(2.1∗2) The mathematical content of type theory is evinced by a certain perspective called *propositions-as-types*, under which one can give the following translations for judgments:

$\Gamma \vdash A\ \mathsf{type}$	"A is a proposition."
$\Gamma \vdash M : A$	"M is a proof of A."
$\Gamma \vdash A = A'$	"A and A' are equal propositions."
$\Gamma \vdash M = M' : A$	"M and M' are equal proofs of propositions."

To emphasize this mathematical interpretation of type theory we will sometimes refer to judgments as *constructions.*

(2.1∗3) The notation $\Gamma \vdash \dots$ signifies the fact that judgments must be stated with respect to a *context* Γ, and $\Gamma\ \mathsf{ctx}$ can be read as Γ is a well-formed context (according to the inference rules). A context can be thought of as the list[1] of assumptions or premises to a proposition or proof. From the empty context, we may iteratively construct larger contexts by means of *context comprehension.* This is captured by the following inference rules:

$$\frac{}{\cdot\ \mathsf{ctx}} \qquad\qquad \frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash A\ \mathsf{type}}{\Gamma, a : A\ \mathsf{ctx}}$$

[1]Not an unordered collection because assumptions may depend on one another.

The point of contexts is to record under what circumstances a construction was carried out. Put another way, contexts reflect the dependency structure of constructions in the sense of **(2∗2)**. The raison d'être of type theory is to provide a rigorous account of constructions that is compatible with variations in the context. This variation is also called a *substitution*, which is a judgment of the form $\sigma : \Gamma \to \Delta$, inductively generated by the following rules:

$$\frac{}{\cdot : \Gamma \to \cdot} \qquad \frac{\sigma : \Gamma \to \Delta \qquad \Gamma \vdash M : A[\sigma]}{\sigma, M/a : \Gamma \to \Delta, a : A}$$

In the above we write $E[\sigma]$ for the operation induced by a substitution on a term or type E in which all occurrences of the indicated variables M/a in σ are replaced by M in E.[2] Contexts and substitutions can be organized into a category. This means that there is an identity substitution $id : \Gamma \to \Gamma$ and one can compose compatible substitutions $\sigma : \Gamma \to \Delta$ and $\delta : \Omega \to \Gamma$ as a single substitution $\delta \circ \sigma : \Omega \to \Delta$. Often we will regard a single term $\Gamma \vdash M : A$ as the substitution $id, M/a : \Gamma \to \Gamma, a : A$ and write $E[M]$ for $E[id, M/a]$.

(2.1∗4) The inference rules regarding specific type connectives come in groups of four: *formation*, *introduction*, *elimination*, and *computation*, which taken together govern how a type is constructed and used. As an example, the *dependent sum* type is governed by the following inference rules:

Σ-FORMATION
$$\frac{\Gamma \vdash A \ \mathsf{type} \qquad \Gamma, a : A \vdash B \ \mathsf{type}}{\Gamma \vdash \Sigma_{a:A}.B(a) \ \mathsf{type}}$$

Σ-INTRODUCTION
$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B[M]}{\Gamma \vdash (M, N) : \Sigma_{a:A}.B(a)}$$

Σ-ELIMINATION-1
$$\frac{\Gamma \vdash P : \Sigma_{a:A}.B(a)}{\Gamma \vdash \mathit{fst}(P) : A}$$

Σ-ELIMINATION-2
$$\frac{\Gamma \vdash P : \Sigma_{a:A}.B(a)}{\Gamma \vdash \mathit{snd}(P) : B[\mathit{fst}(x)]}$$

Σ-COMPUTATION-1
$$\frac{\Gamma \vdash (M, N) : \Sigma_{a:A}.B(a)}{\Gamma \vdash \mathit{fst}(M, N) = M : A}$$

Σ-COMPUTATION-2
$$\frac{\Gamma \vdash (M, N) : \Sigma_{a:A}.B(a)}{\Gamma \vdash \mathit{snd}(M, N) = N : B[M]}$$

The dependent sum is similar to the Cartesian product $A \times B$, except that B is a *type family* that may depend on A. Thus dependent sums subsume the notion of products and we can define $A \times B$ to be $\Sigma_{a:A}.B$.

(2.1∗5) The inference rules **(2.1∗4)** may seem overwhelming at first, but they simply codify the form of dependency discussed in **(2∗2)**. Explicitly, the structure of groups is rendered as the following dependent sum type:

$$\Gamma \vdash \Sigma_{G:\mathcal{U}}.(G \times G \to G) \times G \times (G \to G) \times \dots \ \mathsf{type}$$

We may as well abbreviate this type as *Group*. Moreover, one may formalize group-theoretic formulas/sentences in type theory:

$$\Gamma, G : \mathit{Group}, g : |G| \vdash g \cdot g^{-1} \cdot g = g : |G|,$$

[2]In general the judgment $\Gamma \vdash a : A$ *presupposes* that $\Gamma \vdash A$ type holds; in other words, the former is a sensible statement only in case that the latter is known. Although $A[\sigma]$ cannot be seen *a priori* to be a well-formed type, we may prove after the fact that the operation substitution always induces well-formed types and terms. In such presentations of type theory substitution is called *admissible*. One may also define a version of type theory in which the well-formedness of the substitution operation is *derivable*. For a discussion on the notions of derivability and admissibility, see Harper [52, chp 3].

where we write $|G| : \mathcal{U}$, $\cdot : |G| \times |G| \to |G|$, and $-^{-1} : |G| \to |G|$ for the evident components of G.

(2.1∗6) Considering the reading of **(2.1∗2)**, type theory can be thought of as a structure that mathematizes the notion of proof. But because propositions are merely a *particular* kind of type, this can lead one to consider uninspired "propositions" such as the proposition of being a group **(2.1∗5)**. Since a proposition is just a particular kind of collection (namely a set consisting of *at most* one element, *i.e.* a *subsingleton*), a better intuition is to think about types as an axiomatization of the notion of a *collection*; we will discuss this perspective in Section 2.4.

(2.1∗7) The most important aspect of the algebraic viewpoint is as a tool for studying the *metatheory* of type theories. The idea is that a type theory is defined to be the *free* structure of a specified kind (for instance, one of the structures referenced in **(2.1∗1)**), often referred to as the free model. The free model of a type theory is a construction analogous to other free algebraic structures such as the free group over a set of generators. Crucially, the free model is also equipped with an analogous universal property[3] that allows one to prove properties about the *global properties* of the system of inference rules of a type theory.

The practical consequence of free models and their metatheoretic properties is that they enable us to implement the system of inference rules of type theory as a computer program that mechanically decides whether a certain judgment is derivable (*i.e.* reachable as a node in the derivation). Taken in conjunction with the propositions-as-types perspective, this fact has enabled the development of computer programs called *proof assistants* that allow users to construct formal mathematical proofs that are mechanically verified by the computer. In fact, type theory underlies many state of the art proof assistants that are not only used for formalized mathematics but also program verification (recall that program specifications are just specific kinds of theorems by **(0.3∗1)**).

2.2. THE COMPUTATIONAL PERSPECTIVE

(2.2∗1) The computational perspective in the sense of Martin-Löf [81] can be characterized by the stance that types emerge from (untyped) *computation*; this latter, more primitive notion is often an operational semantics of the form described in **(0.3∗4)**. In this setting, a type is (almost literally) a collection of programs, grouped so together on the basis of their computational behavior. The specifics of how a type describes computational behavior is called a *meaning explanation*. A *computational type theory* is the application of the meaning explanation to a base system of untyped computation.

(2.2∗2) A computational type theory also consists of type judgments similar to the ones of **(2.1∗1)**, except instead of inference rules, type judgments are made evident by appeal to a meaning explanation. To give a meaning explanation of a type A is to give the following information: what is a *canonical* program of A, and when are two such canonical programs are equal.[4] In this context a canonical program is just a terminal or complete computation. The evidence of type judgments consists of mutual interleaving of the meaning explanation for canonical programs and the underlying operational semantics for non-canonical programs.

[3]One can think about this as a (highly sophisticated) induction principle.

[4]Thus a type in computational type theory can be thought of as a kind of Bishop set in which elements range over a given universe of untyped programs.

(2.2∗3) The most significant difference in comparison to the algebraic perspective stems from the semantic and open-ended character of meaning explanations. While a system of rules of the form in **(2.1∗1)** constitutes an inductive definition that fixes the entirety of a type theory from the beginning and the means by which judgments are seen to be derivable, a meaning explanation does not restrict how types and type judgments are established and is compatible with additions to the underlying computation system. The difference is quite stark: a judgment in the algebraic perspective expresses the mechanical derivability, whereas a judgment in the context of the meaning explanation expresses *actual knowledge* about the real world.[5]

One can reconcile the two perspectives by recognizing computational type theory as a particular kind of model of a type theory in the algebraic sense. In this arrangement the mechanical derivation of judgments afforded by an algebraic type theory is guaranteed to be validated by the meaning explanation but is by no means the only way to establish properties about the underlying computational model. For a concrete example, consider the following program

$$f(n) = \mathsf{if}\ \mathsf{hasOne}(\mathit{collatz}(n))\ \mathsf{then}\ 0\ \mathsf{else}\ \text{“}\mathit{bar}''\text{,}$$

where $\mathit{collatz}(n)$ denotes the Collatz sequence and $\mathsf{hasOne}(s)$ is true if and only if $s : \mathbb{N} \to \mathbb{N}$ contains the number one. In a computational type theory one can assign the type $\mathbb{N} \to \mathbb{N}$ to f (if one finds a proof of the Collatz conjecture), but it would be nearly impossible to account for such judgments in the algebraic setting because evidence for their validity may be arbitrarily complex and any attempt to account for them would undermine the mechanical decision of judgments that is critical for building computerized proof assistants.

(2.2∗4) However, the extreme flexibility of the meaning explanation also poses obstacles when trying to design proof assistants based on computational type theories. One can prove lemmas about judgments in a computational type theory analogous to the inference rules of an algebraic type theory to organize routine reasoning and derivations, but often much more engineering is required to make the resulting proof assistant ergonomic for day-to-day use. This and other various theoretical problems with computational type theories has led to a situation in which although proof assistants based on computational type theories such as NuPRL [3] pioneered the concept of type-theoretic proof assistants and many ideas still used today, their usage is somewhat localized in the theorem-proving community in comparison to proof assistants based on algebraic type theories.

2.3. THE SYNTHETIC PERSPECTIVE

(2.3∗1) In this monograph I will mainly use and interact with type theory from a *synthetic* perspective. In contrast to the perspectives outlined in Sections 2.1 and 2.2, the synthetic perspective is not an ontological or philosophical stance about the nature of type theory but rather a position arrived at from the pragmatics of using type theory. Like the computational perspective, one tends to think of types under the synthetic perspective as describing a real phenomenon (in a semantic model). From a model-centric point of view, a type theory is best understood as the *internal language* of the model, that is to say a linguistic way to describe the underyling structures. The benefit of a type-theoretic internal language is *composition*, *i.e.* modularity of construction and reasoning of the kind discussed in **(0.1∗7)**. Compared to the computational perspective, the

[5] In this case untyped computation serves as a notion of reality.

difference is that the models under consideration are not always made up of untyped computation and operational semantics as in **(2.2∗1)**; I outline some examples in Section 2.6.

(2.3∗2) Perhaps the best way to describe the situation is that the objects we require to exist in a type theory can all be given explicit constructions complying with intuitionistic logic; the difference is that the computational character of constructions are not made explicit in terms of operational semantics as in the case of the computational perspective.

This is not to say that we will abandon the computational perspective. To the contrary, we will see that the synthetic perspective gives us the methods to study the layers of computation in the sense of Section 0.5. In particular I show how one can define and study programming languages and their operational semantics *inside* type theory, thence giving us the means to selectively bring out the operational character of programs *internally*, a fact that will become relevant in Chapter 8.

(2.3∗3) The relationship between the synthetic perspective and the algebraic perspective is analogous to one between proof and formalized proof (in some given deductive system). Just as one can carry out informal (but rigorous) mathematical arguments with the understanding that they may be formalized in *e.g.* ZFC, I will carry out the developments in this monograph with the understanding that they may be suitably formalized in a logical framework of the kind discussed in **(2.1∗1)**. Thus the real difference when working in type theory compared to "ordinary mathematics" is the acute awareness of the underlying semantic model, which gives rise to different kinds of internal languages that are suitable for the kind of reasoning required by the given domain of programming and verification. For instance, the domain could be *cost-senstive* programming and verification in the sense of Section 0.3, which would necessitate a different internal language compared to programming and verifying only input-output behaviors.

(2.3∗4) Lastly, a word on metatheory. All of the type theories considered in this monograph are justified by model constructions, *i.e.* they are logically consistent.[6] I do not establish any global metatheoretic properties in the sense of **(2.1∗7)**, which means I have not proven that one can actually build a proof assistant that mechanically checks the validity of the proofs given in terms of these theories. However, this lack of a *formal* justification does not impede us from developing real mechanized case studies by *encoding* them in existing proof assistants, which is discussed in Chapter 4.

2.4. TYPE THEORY AS A MATHEMATICAL UNIVERSE

(2.4∗1) I have hinted at the view of type theory as a framework for developing general mathematics in **(2.1∗2)** and **(2.1∗5)**. To give a sense of the difference between this type-theoretic rendering of mathematics and "ordinary mathematics", I will concentrate on two points that distinguish type theory: the idea that *collections are constructions* and its treatment of dependency.

2.4.1. Collections as constructions.

(2.4.1∗1) Set theory and type theory both aim to give a rigorous account of the pre-mathematical notion of a collection. In type theory a collection is thought of as a *protocol* [7] *of*

[6]Technically, consistent to the relative to the assumed metatheory.

[7]My thanks to Jon for suggesting "protocol" in place of "method", which more accurately describes what I wanted to convey.

construction. For instance, one way to interpret the rule "Σ-introduction" of **(2.1∗4)** as a protocol of construction is as follows. to give a member of the collection $\Sigma x : A.B(a)$ is to give a member $a : A$ of the collection A and a member $b : B(a)$ of the collection $B(a)$. Thus a system of typing rules such as **(2.1∗4)** can be thought of as simultaneously defining a collection and how one is to exhibit elements of the collection. Importantly, to speak of an element one presupposes the collection to which it belongs. Put another way, one can not speak of elements without first identifying its protocol of construction. Notationally, this corresponds to the fact that we always introduce an element $a : A$ along with the way it has been constructed or its associated collection A. Also observe that (up to the equational theory of types) there is a unique type associated to an element, corresponding to the fact that an element can be thought of as the record of an instantiation of the associated protocol of construction.

(2.4.1∗2) In contrast elements in set theory exist independently of their provenance, *i.e.* the protocol by which they were constructed. Whereas elements are constructed following a protocol in type theory, elements are *selected* or *collected* by means of predicates. Consequently elements may belong to multiple sets/collections. The set-theoretic counterpart to $a : A$ is often written as $a \in A$.[8] Practically the difference between $a : A$ and $a \in A$ is that the former is usually completely mechanical, while the latter is a statement of mathematical knowledge that must be justified by proof.

(2.4.1∗3) Set theory is commonly accepted as the foundational basis of modern mathematics, but the use of collections in mathematical practice straddles that of sets and types. While sets are in some ways more flexible than types, they also allow somewhat inane statements such as "$\mathbb{N} \in 5$" that are perfectly sensible according to the canons of set theory. In reality mathematicians often work with the understanding that $\mathbb{N}$ is an *abstract* interface (for instance, as the initial successor algebra or an preordered monoid) that supports operations relevant to their work. Rarely does one rely on the concrete definition of $\mathbb{N}$ so that questions such as "$\mathbb{N} \in 5$" become relevant. The role of type theory is to codify and systematize such uses of abstractions in mathematics.

2.4.2. Dependency in mathematics.

(2.4.2∗1) In contrast to the direct treatment of dependency in type theory, dependency in mathematics is often represented in a more implicit manner that we may well call the "fibred" approach, which counterposes the indexed approach embodied by type theory. To illustrate, take the prototypical dependent type of length-indexed lists of natural numbers $x : \mathbb{N} \vdash \mathsf{vec}(x)\ \mathsf{type}$. The elements of this type are sequences of the specified length, *i.e.* we have $[] : \mathsf{vec}(0)$ and $[1,0,3,2] : \mathsf{vec}(4)$. The explict dependency of vec on the length $n : \mathbb{N}$ may be represented by a non-dependent family as displayed below:

$$\begin{tikzcd}
\mathsf{list} \arrow[d] & l \arrow[d, mapsto] & \{l \in \mathsf{list} \mid |l| = n\} \\
\mathbb{N} & {|l|} & n \arrow[u, mapsto]
\end{tikzcd}$$

[8]There seems to be no persistent and commonly agreed-upon notation that meaningfully distinguishes between the two notions.

The idea is that the types $\mathsf{vec}(n)$ is represented as the *fibres* of the family $\begin{smallmatrix}\mathsf{list}\\ \downarrow\\ \mathbb{N}\end{smallmatrix}$. For every $n : \mathbb{N}$, the inverse image or fibre over n is the set $\{l \in \mathsf{list} \mid |l| = n\}$, which contains exactly the elements corresponding to those that are classified by the type $\mathsf{vec}(n)$. In the fibred view, a dependent element $x : \mathbb{N} \vdash v : \mathsf{vec}(x)$ corresponds to a *section* of the family $\begin{smallmatrix}\mathsf{list}\\ \downarrow\\ \mathbb{N}\end{smallmatrix}$, *i.e.* a function $v : \mathbb{N} \to \mathsf{list}$ such that $|v(n)| = n$.

(2.4.2∗2) The force of the contexts of type theory is implemented by *substitution* or *change of base*: constructions may be transported along a change of contexts. For example, that $\mathsf{vec}(0)$ is a well-formed type corresponds to the fact we may transport the type $\mathsf{vec}(n)$ in the context with one free variable $n : \mathbb{N}$ to the type $\mathsf{vec}(0)$ with no variables along the *closing* substitution $0 : 1 \to \mathbb{N}$ in which 1 represents the empty context and $\mathbb{N}$ represents the singleton context $n : \mathbb{N}$. In the fibred perspective substitution is represented by pullback along the given change of base.[9] For example, we may susbtitute the closing substitution $5 : 1 \to \mathbb{N}$ to obtain the fibre $\mathsf{vec}(5)$:

$$\begin{tikzcd}
len^*5 \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathsf{list} \arrow[d, "len"] \\
1 \arrow[r, "5"'] & \mathbb{N}
\end{tikzcd}$$

Evidently the only fibre of the (trivially indexed) family $\begin{smallmatrix}len^*5\\ \downarrow\\ 1\end{smallmatrix}$ is the set of lists of length 5; (global) sections of this family then correspond exactly to the elements of $\mathsf{vec}(5)$.

(2.4.2∗3) It may seem awkward to express dependency in terms of the fibres of a family at first, but this is common practice in mathematics. For example, one often prefers[10] to work with *fibrations* or *fibred categories* $\begin{smallmatrix}\mathscr{E}\\ \downarrow\\ \mathscr{B}\end{smallmatrix}$ instead of *indexed categories* $\mathscr{B}^{\mathrm{op}} \to \mathbf{Cat}$, which both represent the notion of a *family* of categories $\mathscr{E}_B$ indexed in a base category $\mathscr{B}$, the former from a fibred perspective and the latter from an indexed perspective. Another example would be a *topological vector bundle* $\pi : E \to B$, which formalizes the idea of a family of vector spaces E varying continuously in some base topological space B. Here the fibred perspective is crucial because it is much more natural to state the relationship amongst the fibre vector spaces $E_x = \pi^{-1}(x)$ as fibres of some total space E than to somehow relate disparate spaces as would be required in the indexed perspective.

[9]This is not completely accurate because pullbacks are only defined up to unique isomorphism, but substitutions in type theory are defined up to equality. This well-known problem in the naive semantics of susbtitution as pullbacks can be resolved in many ways (see for instance Hofmann [58]), but this simple picture suffices for our discussion.

[10]See Jacobs [63, sec. 1.10.4] for a deeper discussion.

(2.4.2∗4) Dependent type theory can be thought of as a language that works *implicitly* with respect to variation/substitution/change of base. If the role of a fibration is to explain the semantics of dependency (of categories), then type theory is the language that makes such dependencies routine and easy to reason about.

(2.4.2∗5) Why not just use type theory to reason about every indexed situation? A type theory is a *very* powerful structure: type connectives and operations of the theory always commute with substitutions. From a semantic point of view, this is rarely the case because operations that exist the on *total space* of a fibred category $\begin{smallmatrix}\mathscr{E}\\ \downarrow\\ \mathscr{B}\end{smallmatrix}$ are often not well-behaved with respect to the fibre categories $\mathscr{E}_B$. An example that becomes relevant in Chapter 8 is that of categories of predomains, which are almost always Cartesian closed but not fibre-wise Cartesian closed (in other words, *locally* Cartesian closed).

2.5. TYPE THEORY AS A PROGRAMMING LANGUAGE

(2.5∗1) In addition to its mathematical applications, type theory can also be viewed as a programming language, or better yet, as a language for defining *semantic models* of programs of the kind discussed in Section 0.1. Expanding on the (simply-typed) function signatures of **(0.1∗6)**, type theory brings *dependently-typed* function signatures and functions that enable one to give extremely precise specifications. For example, recalling the example of Euclid's algorithm in **(0.1∗6)**, we may assign the following function signature to *gcd*:

$$\mathit{gcd} : \Pi n, m : \mathbb{N}.\Sigma d : \mathbb{N}.\mathit{GCD}(d, n, m) \tag{2.5∗1∗1}$$

Let me defer explaining the meaning of $\mathit{GCD}(d, n, m)$ for the moment. We have already seen the dependent sum type in **(2.1∗4)**; in this case the type $\Sigma d : \mathbb{N}.\mathit{GCD}(d, n, m)$ classifies pairs $d : \mathbb{N}$ and $p : \mathit{GCD}(d, n, m)$. We have also used the dual *dependent function type* $\Pi a : A.B(a)$ in the above, which classifies functions $f : \Pi a : A.B(a)$ such that $f(a) : B(a)$ for all $a : A$. As for dependent sum types, we may recover ordinary function types $A \to B$ as $\Pi a : A.B$. Let's now consider $\mathit{GCD}(d, n, m)$, which is the following type:

$$((d \mid n) \times (d \mid m)) \times (\Pi d' : \mathbb{N}.((d' \mid n) \times (d' \mid m)) \to d \geq d')$$

The symbol $\mid$ refers to the "divides" relation, so $d \mid n$ means d divides n. What $\mathit{GCD}(d, n, m)$ classifies are tuples $p : (d \mid n)$, $q : (d \mid n)$, and $r : \Pi d' : \mathbb{N}.((d' \mid n) \times (d' \mid m)) \to d \geq d'$, which can be read as (under the propositions as types view) "p is a *proof* that d divides n", "q is a proof that d divides m", and "r is a dependent function such that for all $d' : \mathbb{N}$ dividing both n, m, $r(d')$ is a proof that $d \geq d'$". Thus we have that the property of being a greatest common divisor is expressed as the dependent type $\mathit{GCD}(d, n, m)$. Taken together, this means that Eq. (2.5∗1∗1) expresses *exactly* the desired specification of Euclid's algorithm, namely to compute the greatest common divisor of two natural numbers.

(2.5∗2) There is a slight problem with the example as sketched, which is that the definition given for *gcd* cannot be readily seen as a valid function definition in type theory. The reason was already broached in **(0.2∗2)**: we must scrutinize self-referential or *recursive* definitions to make sure they denote well-defined functions. In **(0.2∗3)** this was used to illustrate the essence of programming languages, which is that programming languages are mechanical disciplines for defining functions. There is nothing wrong with this view, but in practice the situation is more nuanced. Realistic programming languages allow one to write down functions such as $f(x) = f(x+1)$ whose meaning is *divergence*. We call these functions *partial functions* because they are *partially* defined. Similarly, functions as we have assumed so far can be qualified as *total functions* because they are *totally* defined. Under these circumstances, it may seem overly stringent to disallow $f(x) = f(x+1)$ as a valid function.

(2.5∗3) We will eventually discuss the theory of partial functions in Chapter 8; for now we take for granted that a function is a total function. Part of the reason is that type theory is a theory of total functions and partial functions are somewhat of a second-class citizen. The other is that we can get pretty far sans partial functions or divergence. After all, partial functions are not so informative as models of programs — what good is a divergent program in the real world?[11]

(2.5∗4) To enforce the totality of functions, a type theory has rules about how functions are to be defined. In the case of $\mathbb{N} \to \mathbb{N}$ functions, one is typically allowed the discipline of definition by *primitive recursion*: given $b : \mathbb{N}$ and $r : \mathbb{N} \to \mathbb{N}$, there exists a unique function $\mathsf{rec}(b, r) : \mathbb{N} \to \mathbb{N}$[12] satisfying the following:

$$\begin{aligned}&\mathsf{rec}(b, r)(0) = b\\&\mathsf{rec}(b, r)(n+1) = r(\mathsf{rec}(b, r)(n))\end{aligned}$$

More generally, we may present the above in terms of the collection formation, introduction, elimination, and computation rules for the natural numbers $\mathbb{N}$ as we did for dependent sums. The restriction placed on self-reference is that we may only refer to the recursive result associated to the *immedaite predecessor* of the input. The reason for this restriction is that we may prove (by mathematical induction) that every primitive recursive function is well-defined.

Because the definition of *gcd* refers to recursive results associated to numbers other than the immediate predecessor, it cannot be seen as a primitive recursive function in its current form.

(2.5∗5) There are multiple ways to rectify the definition of *gcd* so that it complies with the discipline of primitive recursion. Following the so-called *Bove-Capretta* method [18], one may associate to every candidate function $f : A \to B$ (such as *gcd*) an *accessibility predicate* $\phi : A \to \mathcal{U}$ that can intuitively be thought of as the proof that an argument to the function is well-founded. We can then define a function $f' : \Pi a : A.\phi(a) \to B$ by *structural induction* on the accessibility predicate $\phi(a)$, a discipline of function definition analogous to primitive recursion. We may obtain the original function $f : A \to B$ by proving that all arguments are accessible.

(2.5∗6) In contrast to the Bove-Capretta method, which makes use of inductive families, there is also another more "low-powered" method that has been used in Niu et al. [90] to tame natural patterns of recursion in the context of cost analysis. The idea is that we may abstract the accessibility predicate into an *accessibility bound* that represents the number of recursive calls the function is allowed to make. In the case of *gcd*, we obtain a "truncated" version of Euclid's algorithm as follows.

[11]There are in fact things partiality is good for; for instance self-interpreters [53, Chp 19].

[12]For full generality one should work with *parameterized* functions; here it seemed appropriate to suppress extra parameters for clarity.

$$\begin{aligned}
&\mathit{gcd}_{\clock} : \mathbb{N} \to (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}\\
&\mathit{gcd}_{\clock}(0, n, m) = 0\\
&\mathit{gcd}_{\clock}(k+1, n, 0) = n\\
&\mathit{gcd}_{\clock}(k+1, n, m) = \mathit{gcd}_{\clock}(k, m, n \mathbin{\mathit{mod}} m)
\end{aligned}$$

The expression $\mathit{gcd}_{\clock}$ denotes a primitive recursive function because recursive calls always refer to the immediate predecessor of a fixed number in the input (in this case the abstract accessibility bound k). To get back to the original function, we simply need to find a large enough value for k so that $\mathit{gcd}_{\clock}$ does not "run out" of recursive calls — when $k = 0$, we have that $\mathit{gcd}_{\clock}(k, n, m) = 0$ for any inputs n, m; since zero cannot be said to divide any number, this signals a kind of failure state. In this case, the sum $n + m$ of the inputs suffices to ensure that $\mathit{gcd}_{\clock}$ never fails. More precisely we have the following theorem in type theory:

$$\Pi n, m : \mathbb{N}.\mathit{gcd}(n, m) = \mathit{gcd}_{\clock}(n + m, n, m)$$

Recalling the meaning of the dependent function type, to inhabit the above is to define a function h such that for all $n, m : \mathbb{N}$, $h(n, m)$ is the proof that $\mathit{gcd}(n, m)$ is equal to $\mathit{gcd}_{\clock}(n + m, n, m)$.

***

(2.5*7) Aside from the insistence on totality, type theory is also unique in its treatment of *computational effects* in comparison to most commercially prevalent programming languages. In the case of the latter, a programming language is invariably introduced by means of a "hello world" program:

```
print("hello world")
```

The operational behavior of executing this program involves writing the string "hello world" to a standard output buffer that the user may observe.

What is the semantics of "hello world"? Most programming languages treat the operation of writing to a buffer a *computational effect*; in the absence of further inputs or outputs, "hello world" is then assigned the function signature $1 \to 1$. Thus computational effects are often also referred to as *side effects* to reflect the fact that the effect is not tracked by the program specification, *i.e.* it is something done "off the books". Crucially, a side effect is distinguished from the output of a program, which is done "by the books"; in the case of "hello world" the output is trivial, as indicated by the singleton type 1.

(2.5*8) But something done "off the books" is still semantically significant. This is most salient in the case of *storage effects* in which both reads and writes to the standard output buffer (or other memory buffers) are allowed: a program may read the contents of a buffer and *branch* on this information to produce different *outputs*. Consider a program f with both read and write access to a memory cell `l` storing a natural number:

```
f(n) = n + l
```

This program takes an input number `n` and adds to it the value stored at `l`. Observe that it is semantically unsound to assign the function signature $\mathbb{N} \to \mathbb{N}$ to f: we have not accounted for the contribution of the memory cell `l` to the output!

(2.5∗9) The fact that side effects are not tracked by function signatures in most programming languages has led to and continues to be an endless source of bugs in sufficiently complex software. Mathematically, side effects are an oxymoron — what is the purpose of an "effect" that can be cleanly kept off the books? More objectively, computational effects ought to be thought of as rules codifying a particular pattern of programming. For instance, the pattern of programming with a single storage cell of type S is encoded by extending an ordinary function signature $f : A \to B$ to take into account its presence:

$f : S \times A \to S \times B$

The meaning of the "hello world" program can be expressed by suitably modifying the input store:

$\mathit{hello} : \mathsf{string} \times 1 \to \mathsf{string} \times 1$
$\mathit{hello}(s, u) = (s \frown \text{"hello world"}, u)$

(2.5∗10) Part of the work of the PL community has been to objectify in a similar fashion all manners of semantically dubious notions in extant programming languages that were previously only understood from an ad hoc perspective. In this vein, a goal of this dissertation is to produce a semantically sound model of *cost-sensitive* programming as discussed in Section 0.3; as I will show in Chapter 4, usual conceptions of the cost structure of programs suffer from the problem outlined in **(2.5∗8)**: cost is construed as a side effect that is not quite kept off the books.

2.6. MODELS OF TYPES

(2.6∗1) Confronted with a new mathematical structure or theory, one obtains understanding by relating the new to the old, or something that one already understands. This is called a *model construction*, a process in which one builds a model of a new structure in terms of known concepts. The direction of a model construction is relative — what is known to one may not be known to another, and vice versa. In this monograph I take as known the notion of *sets and functions* and show how one can understand types in terms of these more primitive concepts.

(2.6∗2) I already introduced the idea of modeling dependency in terms of sets and families in Section 2.4.2. Here I will give a more systematic picture of the process by means of a *particular* type theory and its model in sets. This means that I will not explain what it means to be (a model of) a type theory *in general*; the interested reader may find the relevant references in the discussion of **(2.1∗1)**.

(2.6∗3) Recall from **(2.1∗1)** that our goal is to explain the meaning of the following system of inductive definitions:

$\Gamma\ \mathsf{ctx}$
$\Gamma \vdash A\ \mathsf{type}$
$\Gamma \vdash M : A$
$\Gamma \vdash A = A'$
$\Gamma \vdash M = M' : A$

We outline the interpretation of type dependency introduced in Seely [108]. We will associate to every well-formed context Γ ctx a set $\llbracket\Gamma\rrbracket$, to every type-in-context $\Gamma \vdash A$ type a family $\llbracket\Gamma \vdash A\rrbracket$ whose base is $\llbracket\Gamma\rrbracket$ — which we will write as $\begin{array}{c}\llbracket A\rrbracket\\ \downarrow\\ \llbracket\Gamma\rrbracket\end{array}$, and every term-in-context $\Gamma \vdash a : A$ a section of the family $\begin{array}{c}\llbracket A\rrbracket\\ \downarrow\\ \llbracket\Gamma\rrbracket\end{array}$. Equations between types and terms are interpreted as real equalities in the model, *i.e.* if we derive $\Gamma \vdash a = a' : A$ then $\llbracket\Gamma \vdash a : A\rrbracket$ and $\llbracket\Gamma \vdash a' : A\rrbracket$ are equal sections of $\begin{array}{c}\llbracket A\rrbracket\\ \downarrow\\ \llbracket\Gamma\rrbracket\end{array}$.

(2.6∗4) As mentioned in **(2.4.2∗2)**, there is a well-known subtle coherence problem with naively interpreting substitution as pullback that obstructs us from validating equations of the form $\Gamma \vdash A = A'$ in the model as strict equalities; nonetheless I will present the naive model to provide useful intuition. Various solutions to this coherence problem have been found. On the one hand, Hofmann [58] shows how to strictify the semantics to recover the equational theory of substitutions in types, and on the other hand, Curien [28] provides an alternative solution by introducing explicit substitutions. The connection between these two solutions is discussed in Curien, Garner, and Hofmann [29].

(2.6∗5) In **(2.1∗2)** we saw that contexts are generated from the empty context by repeated applications of context comprehension. Semantically we may set the empty context to be the singleton set: $\llbracket\cdot\rrbracket = 1$. Context comprehension corresponds to taking the *total space* of a family. In the situation of a context comprehension $\Gamma, a : A$, we have by the inductive process of the model construction a set $\llbracket\Gamma\rrbracket$ and family $\begin{array}{c}\llbracket A\rrbracket\\ \downarrow\\ \llbracket\Gamma\rrbracket\end{array}$; we then set $\llbracket\Gamma, a : A\rrbracket = \llbracket A\rrbracket$.

(2.6∗6) A substitution $\sigma : \Gamma \to \Delta$ is interpreted as a function $\llbracket\sigma\rrbracket : \llbracket\Gamma\rrbracket \to \llbracket\Delta\rrbracket$, and given $\Delta \vdash A$ type, the type $\Gamma \vdash A[\sigma]$ is given by pulling back the family $\begin{array}{c}\llbracket A\rrbracket\\ \downarrow\\ \llbracket\Delta\rrbracket\end{array}$ along $\llbracket\sigma\rrbracket$, resulting in a family $\begin{array}{c}\llbracket A[\sigma]\rrbracket\\ \downarrow\\ \llbracket\Gamma\rrbracket\end{array}$. The empty substitution $\cdot : \Gamma \to \cdot$ is sent to the unique map into a singleton $\llbracket\Gamma\rrbracket \to 1$. In the situation of an extended substitution $\sigma, M/a : \Gamma \to \Delta, a : A$, we may construct inductively the following pullback.

$$\begin{tikzcd}
\llbracket A[\sigma]\rrbracket \arrow[r, "f"] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & \llbracket A\rrbracket \arrow[d] \\
\llbracket\Gamma\rrbracket \arrow[r, "\llbracket\sigma\rrbracket"'] \arrow[u, bend left, "\llbracket\Gamma \vdash M : A[\sigma]\rrbracket"] & \llbracket\Delta\rrbracket
\end{tikzcd}$$

We may then take the composite $f \circ \llbracket\Gamma \vdash a : A[\sigma]\rrbracket$ as the interpretation of $\sigma, M/a : \Gamma \to \Delta, a : A$.

A similar construction also tells us how to interpret the action of substitution on terms. Given $\Delta \vdash M : A$ and $\sigma : \Gamma \to \Delta$, we construct again a pullback diagram:

$$\begin{tikzcd}
\llbracket A[\sigma] \rrbracket \arrow[r, "f"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \llbracket A \rrbracket \arrow[d] \\
\llbracket \Gamma \rrbracket \arrow[r, "\llbracket \sigma \rrbracket"'] \arrow[u, dotted, bend left, "?"] \arrow[ur, dotted] & \llbracket \Delta \rrbracket \arrow[u, bend right, "\llbracket \Delta \vdash M : A \rrbracket"']
\end{tikzcd}$$

It is a general property of pullback diagrams that the sections indicated are in bijective correspondence to the indicated diagonals factoring the bottom map through the other leg of the pullback. Because $\llbracket \Delta \vdash M : A \rrbracket$ is a section of the family $\begin{array}{c}\llbracket A \rrbracket \\ \downarrow \\ \llbracket \Delta \rrbracket\end{array}$, we may take the diagonal to be $\llbracket \Delta \vdash M : A \rrbracket \circ \llbracket \sigma \rrbracket$.

(2.6∗7) For the type connectives, we can organize our model construction according to the kind of dependency exhibited. There are three cases: dependency is trivial, dependency is propagated, and dependency is basic and nontrivial. We will explain these in turn.

(2.6∗8) First, a type such as $\mathbb{N}$ or 1 are trivially dependent because their semantics are not dependent on the context. We may interpret $\Gamma \vdash \mathbb{N}$ type as the following family:

$$\begin{tikzcd}
\llbracket \Gamma \rrbracket \times \mathbb{N} \arrow[d, "\pi_1"'] \\
\llbracket \Gamma \rrbracket
\end{tikzcd}$$

By an abuse of notation, in the above we also write $\mathbb{N}$ for the set of natural numbers in the model. Observe that sections of $\begin{array}{c}\llbracket \Gamma \rrbracket \times \mathbb{N} \\ \downarrow \\ \llbracket \Gamma \rrbracket\end{array}$ are in bijective correspondence with the set $\mathbb{N}$. Similarly, the singleton type $\Gamma \vdash 1$ type is interpreted as the identity family $\begin{array}{c}\llbracket \Gamma \rrbracket \times 1 \\ \downarrow \\ \llbracket \Gamma \rrbracket\end{array} \cong \begin{array}{c}\llbracket \Gamma \rrbracket \\ \downarrow \\ \llbracket \Gamma \rrbracket\end{array}$.

(2.6∗9) In the second case a compound type connective may simply "pass along" the source of dependency arising from its constituent types. Consider the dependent sum type

$$\begin{array}{l}\Sigma\text{-FORMATION}\\ \dfrac{\Gamma \vdash A \ \mathsf{type} \qquad \Gamma, a : A \vdash B \ \mathsf{type}}{\Gamma \vdash \Sigma_{a:A}.B(a) \ \mathsf{type}}\end{array}$$

In this situation, we have inductively constructed the families $\begin{array}{c}\llbracket A \rrbracket \\ \downarrow \\ \llbracket \Gamma \rrbracket\end{array}$ and $\begin{array}{c}\llbracket B \rrbracket \\ \downarrow \\ \llbracket A \rrbracket\end{array}$. We need to construct a family whose base is $[\![\Gamma]\!]$: this is given by postcomposing $\begin{smallmatrix}[\![B]\!]\\ \downarrow\\ [\![A]\!]\end{smallmatrix}$ with $\begin{smallmatrix}[\![A]\!]\\ \downarrow\\ [\![\Gamma]\!]\end{smallmatrix}$. For example, we can compute the meaning of the type $\Gamma \vdash \Sigma_{n:\mathbb{N}}.\mathsf{vec}(n)$ in which $\Gamma, n : \mathbb{N} \vdash \mathsf{vec}\ \mathsf{type}$ is interpreted by the family given in **(2.4.2∗1)**:

$$\begin{tikzcd}
{[\![\Gamma]\!] \times \mathsf{list}} \arrow[d, "id \times |-|"'] \\
{[\![\Gamma]\!] \times \mathbb{N}} \arrow[d, "\pi_1"'] \\
{[\![\Gamma]\!]}
\end{tikzcd}$$

Sections of this family are in bijective correspondence with the lists of natural numbers, reflecting the isomorphism $(\Sigma n : \mathbb{N}.\mathsf{vec}(n)) \cong \mathsf{list}$.

(2.6∗10) Consider now the introduction rule:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B(a)}{\Gamma \vdash (a, b) : \Sigma_{a:A}.B(a)}\ \textsc{Σ-introduction}$$

Semantically we have the following situation:

$$\begin{tikzcd}
{[\![B(a)]\!]} \arrow[r, "f"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & {[\![B]\!]} \arrow[d] \\
{[\![\Gamma]\!]} \arrow[r, "{[\![\Gamma \vdash a : A]\!]}"'] \arrow[u, bend left, "{[\![\Gamma \vdash b : B(a)]\!]}"] \arrow[ur, dotted] & {[\![A]\!]} \arrow[d] \\
 & {[\![\Gamma]\!]} \arrow[u, bend left, "{[\![\Gamma \vdash a : A]\!]}"] \arrow[uu, dotted, bend right, "?"']
\end{tikzcd}$$

We are to find a section of $\begin{smallmatrix}[\![B]\!]\\ \downarrow\\ [\![\Gamma]\!]\end{smallmatrix}$ as indicated above. By an argument similar to the one in **(2.6∗6)**, Because $[\![\Gamma \vdash a : A]\!]$ is a section of $\begin{smallmatrix}[\![A]\!]\\ \downarrow\\ [\![\Gamma]\!]\end{smallmatrix}$, it suffices to find a map $[\![\Gamma]\!] \to [\![B]\!]$ factoring $[\![\Gamma \vdash a : A]\!]$ through $\begin{smallmatrix}[\![B]\!]\\ \downarrow\\ [\![A]\!]\end{smallmatrix}$, as indicated by the dotted diagonal. By an argument similar to the one in **(2.6∗6)**, we may take this to be $f \circ [\![\Gamma \vdash b : B(a)]\!]$ since $[\![\Gamma \vdash b : B(a)]\!]$ is a section.

By similar arguments we may also interpret the elimination and computation rules of the dependent sum type.

(2.6∗11) Another example of a type connective exhibiting basic nontrivial dependency is the (extensional) equality type, typically characterized by the following rules:

$$\textsf{eq}\text{-FORMATION}\quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash \textsf{eq}_A(a,b)\ \textsf{type}} \qquad \textsf{eq}\text{-INTRODUCTION}\quad \frac{\Gamma \vdash a = b : A}{\Gamma \vdash \textsf{refl} : \textsf{eq}_A(a,b)} \qquad \textsf{eq}\text{-ELIMINATION}\quad \frac{\Gamma \vdash u : \textsf{eq}_A(a,b)}{\Gamma \vdash a = b : A}$$

$$\textsf{eq}\text{-UNIQUENESS}\quad \frac{\Gamma \vdash u : \textsf{eq}_A(a,b) \qquad \Gamma \vdash v : \textsf{eq}_A(a,b)}{\Gamma \vdash u = v : \textsf{eq}_A(a,b)}$$

The purpose of the equality type is to *internalize* equality of terms at the judgment level so that one may use equalities as premises to constructions.

(2.6∗12) The equality type $\Gamma \vdash \textsf{eq}_A(a,b)$ $\textsf{type}$ is interpreted as the equalizer of the interpretation of the terms $\Gamma \vdash a, b : A$, displayed horizontally below:

$$\begin{tikzcd}
{[\![A]\!]} \arrow[d] & \\
{[\![\Gamma]\!]} \arrow[u, bend left=60, "{[\![\Gamma \vdash a : A]\!]}"] \arrow[u, bend right=60, "{[\![\Gamma \vdash b : A]\!]}"'] & {[\![\textsf{eq}_A(a,b)]\!] = \{x \in [\![\Gamma]\!] \mid [\![\Gamma \vdash a : A]\!](x) = [\![\Gamma \vdash b : A]\!](x)\}} \arrow[l, hook']
\end{tikzcd}$$

Observe that when a and b are equal terms we have that $[\![\Gamma \vdash a : A]\!]$ and $[\![\Gamma \vdash b : A]\!]$ are equal sections of $\begin{array}{c}[\![A]\!]\\ \downarrow\\ [\![\Gamma]\!]\end{array}$ and so $\begin{array}{c}[\![\textsf{eq}_A(a,b)]\!]\\ \downarrow\\ [\![\Gamma]\!]\end{array}$ is just the identity family $\begin{array}{c}[\![\Gamma]\!]\\ \downarrow\\ [\![\Gamma]\!]\end{array}$, which as we noted in **(2.6∗8)** is the interpretation of the singleton type $\Gamma \vdash 1$ $\textsf{type}$.

(2.6∗13) The construction outlined here extends to structures other than sets and functions; what is needed in general is a *locally Cartesian closed category* or lccc. The interested reader may refer to Jacobs [63, chp. 1] for an introduction to lccc's and Seely [108] and Hofmann [58] for the interpretation of type theory in lccc's.

* * *

(2.6∗14) Once we have established the type-theoretic structure of sets, we can use type theory as the "internal language" of sets, *i.e.* a convenient way to reason about sets and functions. This theory-model relationship is mutually beneficial: the model gives mathematical justification for the theory, and conversely, the theory gives us a good way to reason about the model. In this way, we can be sure that we can use type theory to prove something real: for instance, we can unfold a type-theoretic proof of the correctness of Euclid's algorithm as defined in **(2.5∗1)** into an ordinary mathematical proof about Euclid's algorithm on the "real" natural numbers.

CHAPTER 3

Variable sets

3.1. PRESHEAVES

(3.1∗1) The primary source of models of types we will consider come from categories of presheaves. This section serves as an overview of some important properties of presheaves.

(3.1∗2) Given a small category $\mathscr{C}$, a *presheaf* on $\mathscr{C}$ is a contravariant functor $\mathscr{C}^{\mathrm{op}} \to \mathbf{Set}$. We write $\widehat{\mathscr{C}}$ for the category of presheaves on $\mathscr{C}$ and natural transformations.

(3.1∗3) It is common to think about presheaves as *variable sets.* For instance, a presheaf F on the lattice of open sets of a topological space X (considered as a posetal category) assigns to every open set U a set $F(U)$ that is compatible with the poset structure of open sets: for every inclusion $U \subseteq V$, we have an induced *restriction* $F(V) \to F(U)$ such that the restrictions $F(W) \to F(V) \to F(U)$ and $F(W) \to F(U)$ induced by $U \subseteq V \subseteq W$ and $U \subseteq W$ are equal. For example, we can define the following presheaf C:

$$C(U) = \{f : U \to \mathbb{R} \mid f \text{ continuous}\}$$

Restrictions $C(V) \to C(U)$ in this case are given by function restrictions.

(3.1∗4) An example that is central to this dissertation are the category of presheaves on the interval poset $\mathbb{I} = [1] = \{0 \sqsubseteq 1\}$. A presheaf F on $\mathbb{I}$ is a family $\begin{array}{c} F(1) \\ \downarrow \\ F(0) \end{array}$. We sometimes write $\mathbf{Set}^{\to}$ for this presheaf category.

(3.1∗5) If the model of types in **Set** (as sketched in Section 2.6) is meant to capture "ordinary mathematics" in type theory, then a model of types in a category of presheaves $\widehat{\mathscr{C}}$ can be thought of as a mathematical universe that is variable in an *abstract world structure* given by the base category $\mathscr{C}$. One way to think about the situation is that a presheaf is a mathematical structure to attach data to each "world" in $\mathscr{C}$ in a way that is compatible with the world structure, which is encoded by the restriction action and its associated laws.

3.1.1. Representing cost structure in presheaves.

(3.1.1∗1) Recall from Section 0.3 that our goal is to integrate the cost structure of programs with ordinary functional semantics so that we can both make sense of cost-sensitive properties (such as resource usage characteristics) and pure functional properties (such as input-output specifications) in a single theory. The relevance of presheaves is made clear by observing that we can represent these two viewpoints by means of the world structure $\mathbb{I}$ in which the point "0" represents the "functional world" and the point "1" represents the cost-sensitive world, and the relation $0 \sqsubseteq 1$ represents the fact that we have a way to *redact* or *seal away* cost information (remember that the restriction goes in the other direction). Thus we can think of $\widehat{\mathbb{I}}$ as a mathematical setting in which to speak coherently about the cost structure of programs while retaining the ability to reason about them *qua* functions when needed.

(3.1.1∗2) Concretely, one should think about a presheaf $\begin{matrix} X(1) \\ \downarrow \\ X(0) \end{matrix}$ as encoding a family of cost-sensitive functions that projects the underlying pure function. For instance, given a monoid $\mathbb{C}$ representing program cost, we have a family $\begin{matrix} \mathsf{list} \times \mathbb{C} \\ \downarrow \\ \mathsf{list} \end{matrix}$ defined by the projection map π_1. As mentioned in **(0.3∗5)**, the total space $\mathsf{list} \times \mathbb{C}$ represents the set of cost-sensitive programs of type list that are sent to their functional component by the projection. Naturally, a cost-sensitive *function* is a map of presheaves $X \to Y$, *i.e.* a natural transformation.

(3.1.1∗3) The category of presheaves over $\mathbb{I}$ integrates the purely functional semantics and cost-sensitive semantics that are *independently* represented in terms of the base and total space of families of the form in **(3.1.1∗2)**. Moreover, this integration is coherent in the following sense: *cost structure cannot interfere with functional semantics.* This immediately follows by unfolding the definition of a cost-sensitive function f:

$$\begin{tikzcd}
X(1) \arrow[r, "f_1"] \arrow[d, "X"'] & Y(1) \arrow[d, "Y"] \\
X(0) \arrow[r, "f_0"'] & Y(0)
\end{tikzcd}$$

Thinking of the families X and Y as defined by projections sending a cost-sensitive element to its functional component, we see that given a *fixed* functional component $x \in X(0)$, any cost-sensitive inputs u, v in the fibre over x must be sent to cost-sensitive outputs whose functional component is completely determined by x. In other words, $f_1(u)$ and $f_1(v)$ must have the same functional component.

(3.1.1∗4) Besides cost structure, presheaves are used to give mathematical semantics to a wide range of programming features and type theories. Applications include *concurrency/higher-dimensional automata* [33], *guarded recursion/guarded type theory* [13, 16, 120], *cubical type theory* [12, 61, 5], and *normalization proofs* [27, 118].

3.1.2. The Yoneda lemma.

(3.1.2∗1) Another perspective on presheaves is that they provide an a rich internal language that one can use to reason about external structures that otherwise do not support an "intrinsic" logical language. Concretely, the process of taking presheaves on a small category $\mathscr{C}$ yields a functor $\mathsf{y}_{\mathscr{C}} : \mathscr{C} \to \widehat{\mathscr{C}}$ sending an object C to the *representable presheaf* $\mathsf{y}C : \widehat{\mathscr{C}}$, whose action on objects is defined as follows.

$$\begin{array}{l} \mathsf{y}C : \mathscr{C}^{\mathrm{op}} \to \mathbf{Set} \\ (\mathsf{y}C)D = \mathrm{Hom}_{\mathscr{C}}(D, C) \end{array}$$

Given $f : D' \to D$, the restriction action of the representable presheaf $(\mathsf{y}C)f : \mathrm{Hom}(D, C) \to \mathrm{Hom}(D', C)$ is given by precomposition. On morphisms, y sends $g : C \to C'$ to the natural transformation $\mathsf{y}C \to \mathsf{y}C'$ whose component at D is given by postcomposition. In other words we have the following naturality square for every $D' \to D$,

$$\begin{tikzcd}
\mathrm{Hom}(D, C) \arrow[r, "g \circ -"] \arrow[d, "- \circ f"'] & \mathrm{Hom}(D, C') \arrow[d, "- \circ f"] \\
\mathrm{Hom}(D', C) \arrow[r, "g \circ -"'] & \mathrm{Hom}(D', C')
\end{tikzcd}$$

The naturality square above follows because function composition is associative.

(3.1.2∗2) Presheaves in the image of y are called representable presheaves because an arbitrary presheaf F can be reconstructed (up to natural isomorphism) in terms of maps out of representable presheaves, in the sense that the following isomorphism holds:

$$\mathrm{Hom}_{\widehat{\mathscr{C}}}(\mathsf{y}C, F) \cong F(C)$$

naturally in both F and C. This means that for every natural transformation $F \to G$ we have a naturality square:

$$\begin{tikzcd}
\mathrm{Hom}(\mathsf{y}C, F) \arrow[r] \arrow[d] & F(C) \arrow[d] \\
\mathrm{Hom}(\mathsf{y}C, G) \arrow[r] & G(C)
\end{tikzcd} \tag{3.1.2∗2∗1}$$

and a similar square for every morphism $C \to D$. This property is known as the *Yoneda lemma*; it is a somewhat "obvious" property about presheaf categories that has profound implications when studying the semantics of type theories.

(3.1.2∗3) A simple corollary of the Yoneda lemma is that the functor $\mathsf{y} : \mathscr{C} \to \widehat{\mathscr{C}}$ is fully faithful; consequently we often refer to y as the *Yoneda embedding.* This fact enables a common technique by which one studies a poorly structured category $\mathscr{C}$ (from a logical perspective) through its yoneda embedding $\widehat{\mathscr{C}}$, which supports a very expressive internal type theory. One can then transport theorems about representable objects in the presheaf category back to the original category via the embedding. For example, this kind of reasoning has been used to establish metatheorems about the syntax of type theory [118, 116].

(3.1.2∗4) The Yoneda embedding has also been used to construct models of *synthetic domain theory* [62], which is a way of incorporating general recursion and partiality into type theory (recall from our discussion in Section 2.5 that type theory only supports total functions by default). In Chapter 7 we use (internal) presheaf categories to provide a *cost-sensitive* model of synthetic domain theory.

✎ **(3.1.2∗5)** As a concrete application of the Yoneda lemma, we may prove a characterization of monomorphisms in presheaf categories. Recall that a monomorphism is a left cancellable map, *i.e.* a map $m : F \rightarrowtail G$ such that for all $f, g : X \to F$ with $mf = mg$, we have $f = g$. We may give a concrete description of a monomorphism $m : F \rightarrowtail G$ in a presheaf category $\widehat{\mathscr{C}}$ in terms of monomorphisms in **Set**, *i.e.* injective functions: a map of presheaves $m : F \to G$ is mono if and only if $m_C : F(C) \to G(C)$ is injective for all $C : \mathscr{C}$.

■ **(3.1.2∗6)** Suppose that $m : F \to G$ is mono in $\widehat{\mathscr{C}}$. To show that $m_C : F(C) \to G(C)$ is injective, fix $x, y \in F(C)$ with $m_C(x) = m_C(y)$. We want to show that $x = y$. Writing $\overline{x} : \mathsf{y}C \to F$ for the map of presheaves induced by an element $x \in F(c)$, we have the following configuration of maps in $\widehat{\mathscr{C}}$:

$$\begin{tikzcd}
\mathsf{y}C \arrow[r, bend left, "\overline{x}"] \arrow[r, bend right, "\overline{y}"'] & F \arrow[r, "m"] & G
\end{tikzcd}$$

By **(3.1.2∗2)**, we have that $\overline{-} : F(C) \cong \mathrm{Hom}(\mathsf{y}C, F)$ is a bijection, so it suffices to show that $\overline{x} = \overline{y}$. Moreover, because m is mono, we just need to show that $m\overline{x} = m\overline{y}$. Observing by assumption that $m_C(x) = m_C(y)$ induces equal maps $\overline{m_C(x)} = \overline{m_C(y)} : \mathsf{y}C \to G$, it suffices to show that $m\overline{x} = \overline{m_C(x)}$ and $m\overline{y} = \overline{m_C(y)}$, *i.e.* $\overline{-}$ commutes with m. But this is exactly the naturality square Eq. (3.1.2∗2∗1). The reader is invited to check the other direction of the claim.

3.1.3. Types in presheaves.

(3.1.3∗1) One can also give an interpretation of types in terms of presheaves much like the model of types in sets outlined in Section 2.6. The main intuition is that compound types are defined in a pointwise fashion. In this section we show that presheaf categories are Cartesian closed; proofs of local Cartesian closure can be found in Awodey [8, p. 236], and models of dependent types in presheaves can be found in Jacobs [63, S 10.5.9] and Awodey and Rabe [10].

(3.1.3∗2) In the following, fix a small category $\mathscr{C}$. As a simple example, we have that the unit type/terminal object of $\widehat{\mathscr{C}}$ is the constant presheaf $1_{\widehat{\mathscr{C}}} : \mathscr{C}^{\mathrm{op}} \to \mathbf{Set}$ determined (up to unique isomorphism) by the terminal object $1_{\mathbf{Set}} = \{*\}$ of the category of sets.

(3.1.3∗3) The Cartesian product of two presheaves F, G is defined componentwise in terms of products in **Set**: $(F \times G)(C) = F(C) \times G(C)$. The fact that this defines a product follows because $F(C) \times G(C)$ is a product in **Set**. In fact, $\widehat{\mathscr{C}}$ is closed under all small[1] limits and colimits, which

[1]A categorical gadget is called *small* when it can be seen as an object of **Set**, *i.e.* a small category has a *set* of objects and hom-*sets*. A small (co)limit is one in which the associated diagram is indexed by a small category.

are computed in a similar pointwise fashion. For example, we have that the natural numbers type $\mathbb{N}_{\widehat{\mathscr{C}}}$ in $\widehat{\mathscr{C}}$ is the constant presheaf determined by the actual natural numbers $\mathbb{N}$ in **Set**, since we may compute that $\mathbb{N}_{\widehat{\mathscr{C}}}(C) = (\amalg_{n\in\mathbb{N}} 1_{\widehat{\mathscr{C}}})(C) = \amalg_{n\in\mathbb{N}}((1_{\widehat{\mathscr{C}}})(C)) = \amalg_{n\in\mathbb{N}} 1_{\mathbf{Set}} = \mathbb{N}$.

(3.1.3∗4) As another application of the Yoneda lemma, we will give a formula for exponential objects in presheaf categories. Recall that the exponential object in a category is determined as the right adjoint in the product-hom adjunction $(- \times B) \dashv -^B$, giving rise to the following natural isomorphism:

$$\mathrm{Hom}(A \times B, C) \cong \mathrm{Hom}(A, C^B) \tag{3.1.3∗4∗1}$$

Using Eq. (3.1.3∗4∗1), we may derive what the exponential object in $\widehat{\mathscr{C}}$ *must be*, if it exists:

$$\begin{aligned} G^F(C) &\cong \mathrm{Hom}(\mathsf{y}C, G^F) && \text{Yoneda} \\ &\cong \mathrm{Hom}(\mathsf{y}C \times F, G) && ((- \times B) \dashv -^B) \end{aligned}$$

In the case that $\mathscr{C}$ is a linear order with relations of the form $C_n \sqsubseteq C_{n+1}$, we may visualize presheaves F, G as sets varying over a horizontal axis and identify the component of the exponential G^F at C as the following natural family of vertical maps for every $C_n \sqsubseteq C$:

$$\begin{tikzcd} \ldots \arrow[r] & F(C_{n+1}) \arrow[r] \arrow[d] & F(C_n) \arrow[r] \arrow[d] & \ldots \\ \ldots \arrow[r] & G(C_{n+1}) \arrow[r] & G(C_n) \arrow[r] & \ldots \end{tikzcd} \tag{3.1.3∗4∗2}$$

Thus one can think of $G^F(C)$ as the set of C-"bounded" natural transformations $F \to G$. The restriction action of G^F is given by postcomposition. In this example, given $D \sqsubseteq C$, the family displayed in Eq. (3.1.3∗4∗2) is restricted to consist of the vertical maps $F(D_i) \to G(D_i)$ for $D_i \sqsubseteq D \sqsubseteq C$.

(3.1.3∗5) A useful fact we will often rely on is that the Yoneda embedding preserves any limit that exists in the base category. In particular, this implies that $\mathscr{C} \to \widehat{\mathscr{C}}$ is a Cartesian closed functor, *i.e.* exponential objects are preserved:

$$\begin{aligned} \mathsf{y}(Y^X)(C) &= \mathrm{Hom}_{\mathscr{C}}(C, Y^X) \\ &\cong \mathrm{Hom}_{\mathscr{C}}(C \times X, Y) && \text{(adjointness)} \\ &\cong \mathrm{Hom}_{\widehat{\mathscr{C}}}(\mathsf{y}(C \times X), \mathsf{y}Y) && \text{(y full and faithful)} \\ &\cong \mathrm{Hom}_{\widehat{\mathscr{C}}}(\mathsf{y}C \times \mathsf{y}X, \mathsf{y}Y) && \text{(y preserves limits)} \\ &\cong \mathrm{Hom}_{\widehat{\mathscr{C}}}(\mathsf{y}C, \mathsf{y}Y^{\mathsf{y}X}) && \text{(adjointness)} \\ &= \mathsf{y}Y^{\mathsf{y}X} && \text{(definition)} \end{aligned}$$

Naturality follows because all isomorphisms in the above are natural.

3.1.4. Density.

(3.1.4∗1) The fact that the Yoneda embedding $\mathscr{C} \to \widehat{\mathscr{C}}$ is full and faithful can also be understood geometrically. In particular we have that every presheaf $X : \widehat{\mathscr{C}}$ is given by a *canonical* colimit of representable presheaves, a property more generally referred to as *density* and equivalent to the full and faithfulness of Yoneda. This alternative characterization will be a useful technical device when constructing models of type theories based on the Yoneda embedding (for instance we employ this fact when constructing models of *synthetic domain theory* in Chapter 7; see in particular **(7.4∗8)**).

📖 **(3.1.4∗2)** Given functors $F : \mathscr{D} \to \mathscr{C}$ and $G : \mathscr{E} \to \mathscr{C}$, define the *comma category* $F \downarrow G$ as follows. The objects of $F \downarrow G$ consist of triples $(D : \mathscr{D}, E : \mathscr{E}, F(D) \xrightarrow{\alpha} G(E))$ and the arrows are morphisms $f : D \to D'$ and $g : E \to E'$ such that the following diagram in $\mathscr{C}$ commutes:

$$\begin{tikzcd}
F(D) \arrow[r, "\alpha"] \arrow[d, "Ff"'] & G(E) \arrow[d, "Gg"] \\
F(D') \arrow[r, "\alpha'"'] & G(E')
\end{tikzcd}$$

When either F or G is a constant functor we use the determining object to denote the corresponding functor. For instance, we write $F \downarrow C$ when $G : \mathscr{C} \to \mathscr{C}$ is the constant functor determined by $C : \mathscr{C}$. There are two evident projection functors $\pi_{\mathscr{D}} : F \downarrow G \to \mathscr{D}$ and $\pi_{\mathscr{E}} : F \downarrow G \to \mathscr{E}$.

(3.1.4∗3) Fix a presheaf $F : \mathscr{C}^{\mathrm{op}} \to \mathbf{Set}$. We seek to reconstruct F as a canonical colimit of representables. More precisely, let $\mathrm{Elts}(F)$ by the *category of elements* of F, defined to be the comma category $\mathsf{y}_{\mathscr{C}} \downarrow F$. Explicitly, the objects of $\mathrm{Elts}(F)$ consists of pairs (C, p) with $C : \mathscr{C}$ and $p \in F(C)$, and a morphism $(C, p) \to (C', p')$ is a map $f : C \to C'$ in $\mathscr{C}$ such that the restriction of p' along f is p. We have a small diagram $J \to \mathscr{C}$ given by the forgetful functor $\mathrm{Elts}(F) \to \mathscr{C}$. Then the density of Yoneda is the statement that F is naturally isomorphic to the colimit of the diagram $J \to \mathscr{C} \hookrightarrow \widehat{\mathscr{C}}$.

📖 **(3.1.4∗4)** More generally, we may consider the the situation in which a functor $i : \mathscr{D} \to \mathscr{C}$ (usually a subcategory inclusion) is *dense* in the following sense: every $C \in \mathscr{C}$ is isomorphic to the colimit $\mathrm{Colim}(i \downarrow C \xrightarrow{\pi_{\mathscr{D}}} \mathscr{D} \xrightarrow{i} \mathscr{C})$. Observe that the density of Yoneda is a special case in which i is $\mathsf{y} : \mathscr{C} \to \widehat{\mathscr{C}}$.

(3.1.4∗5) There is also a corresponding generalization of the Yoneda embedding for dense functors: a functor $\mathscr{D} \to \mathscr{C}$ is dense if and only if the *nerve*/restricted Yoneda embedding $N : \mathscr{C} \to [\mathscr{D}^{\mathrm{op}}, \mathbf{Set}]$ is full and faithful.

3.2. INTERNAL LANGUAGE OF PRESHEAF CATEGORIES

(3.2∗1) Every presheaf category supports a rich internal language that one can use to develop mathematics that “varies” over the base category. The relevant variation for this dissertation is the variation between the cost-sensitive and functional world as discussed in **(3.1.1∗1)**. In this section we outline the logical aspect of this internal language by means of the *Kripke-Joyal semantics* of presheaf categories [78].

3.2.1. Propositions in presheaf categories.

(3.2.1∗1) In ordinary mathematics (*i.e.* mathematics in the internal language of **Set**), a predicate over a set X may be identified as either a subset $S \subseteq X$ or a characteristic function $X \to 2$, where 2 is any two element set; conventionally we write $2 = \{\bot, \top\}$. We may refer to the function $\top : 1 \to 2$ determined by the element $\top$ as the *subset classifier* of **Set**, because every subset $S \subseteq X$ fits into a pullback diagram for a unique characteristic map $X \to 2$ as follows.

$$\begin{tikzcd} S \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & 1 \arrow[d, "\top"] \\ X \arrow[r] & 2 \end{tikzcd} \qquad (3.2.1{*}1{*}1)$$

Namely, $X \to 2$ sends $x \in X$ to $\top$ if and only if $x \in S$.

(3.2.1∗2) In an arbitrary category subsets may be replaced by *subobjects*, and a category with finite limits is said to have a *subobject classifiers* when there is an object Ω and a map $\top : 1 \to \Omega$ such that every monomorphism $S \rightarrowtail X$ fits into a pullback diagram as in Eq. (3.1.3∗4∗2).

(3.2.1∗3) To gain some intuition about mathematics in the language of a presheaf category $\widehat{\mathscr{C}}$, it is instructive to look at how the subobject classifier is defined in $\widehat{\mathscr{C}}$.

(3.2.1∗4) A *sieve* on an object $C : \mathscr{C}$ is a collection of arrows S with codomain C closed under precomposition. Equivalently, a sieve on C is a monomorphism $S \rightarrowtail \mathsf{y}C$ in $\widehat{\mathscr{C}}$. We write $\mathsf{Sieve}(C)$ for the set of sieves on C. The total sieve on C is defined as $\mathsf{y}C$ itself, which we write as t_C.

(3.2.1∗5) Define the presheaf Ω by sending C to $\mathsf{Sieve}(\mathrm{C})$. Given $f : C \to D$, the restriction action $\mathsf{Sieve}(D) \to \mathsf{Sieve}(C)$ is given by precomposition: given a sieve T on D, define a sieve on S on C by $g \in S$ if and only if $fg \in T$. The subobject classifier of $\widehat{\mathscr{C}}$ is the map $\top : 1 \to \Omega$ whose components $\top_C \in \Omega(C)$ are defined as the corresponding total sieves.

3.2.1.1. Propositions in the interval presheaf category.

(3.2.1.1∗1) Unfolding **(3.2.1∗5)** in the presheaf category $\widehat{\mathbb{I}} = \widehat{\{0 \sqsubseteq 1\}} = \mathbf{Set}^{\to}$, we have a family Ω as displayed below:

$$\begin{tikzcd} \{\emptyset, I, t\} \arrow[d] \\ \{\emptyset, t\} \end{tikzcd} \qquad (3.2.1.1{*}1{*}1)$$

In the above, $\emptyset$ and t are the empty sieve and total sieves, respectively, and I is the sieve on 1 consisting of just the arrow $0 \sqsubseteq 1$. the action of the presheaf Ω sends the empty and total sieves to themselves and I to the total sieve.

(3.2.1.1∗2) Thinking of a sieve on C as a truth value at the world C, the family Eq. (3.2.1.1∗1∗1) gives a visual representation of the variable logical structure of $\widehat{\mathbb{I}}$. On the one hand, the world at

$C = 0$ corresponding to the base $\{\emptyset, t\}$ is the world of ordinary propositions in which there are only two truth values. On the other hand, the world at $C = 1$ corresponding to the total space $\{\emptyset, I, t\}$ contains an "intermediate" or indeterminate truth value that becomes true in the world at $C = 0$. As I will show in Chapter 4, this configuration of logical worlds is precisely the structure needed to capture the variation between the cost-sensitive ($= 1$) and functional ($= 0$) worlds in which the intermediate truth value is used to classify cost-sensitive data that becomes stripped away in the functional world.

(3.2.1.1∗3) Consequently, there are three global elements $1 \to \Omega$ in $\widehat{\mathbb{I}}$: the constantly true proposition $\top$, the constantly false proposition $\bot$, and the variable/intermediate proposition u determined by the intermediate truth value I.

(3.2.1.1∗4) Likewise, a subobject $S \rightarrowtail X$ is determined by a square as follows.

$$\begin{tikzcd} X(1) \arrow[r] \arrow[d] & \{\emptyset, I, t\} \arrow[d] \\ X(0) \arrow[r] & \{\emptyset, t\} \end{tikzcd}$$

Recalling from **(3.1.1∗2)** the example of encoding cost structure as a presheaf X over $\mathbb{I}$, the existence of an intermediate truth value allows us to define *cost-sensitive* properties that *restricts* to ordinary functional predicates. For instance, we may define the proposition of cost-sensitive equality $x =_X y$ that restricts to ordinary equality on the functional components of x and y:

$$\begin{tikzcd} X(1) \times X(1) \arrow[r] \arrow[d] & \{\emptyset, I, t\} \arrow[d] \\ X(0) \times X(0) \arrow[r] & \{\emptyset, t\} \end{tikzcd}$$

The bottom map is the characteristic map of ordinary equality, and the top map is defined as follows.

$$(x, y) \mapsto \begin{cases} t & \text{if } x = y \\ I & \text{if } (x \neq y) \wedge \exists[z]\ x, y \in (X \times X)_{z,z} \\ \emptyset & \text{o.w.} \end{cases}$$

The idea is that if x and y are not equal but are both in the fibre $(X \times X)_{z,z}$ of a single functional component z then the proposition $x = y$ is in an "intermediate" state and only becomes true in the functional world.

3.2.2. Kripke-Joyal semantics.

(3.2.2∗1) So far we have the logical structure of presheaf categories from an *external* perspective typical of ordinary mathematical developments. In contrast, one may also work in the *internal* mathematics of presheaf categories, which is in general higher-order intuitionistic logic that may be extended based on the structure of the base category. The connection between the internal mathematics of presheaf categories and external mathematics may be explicated by means of the associated *Kripke-Joyal semantics* [78]. While traditionally presented in the language of higher-order logic, one can also work relative to an analogous internal *type theory* whose connection to external mathematics is explained by a *proof-relevant* Kripke-Joyal semantics [9].

(3.2.2∗2) Internal mathematics is type-theoretic in the sense that context management is kept implicit. However, because external mathematics keeps track of contexts explicitly, we shall mediate between internal and external mathematics via the notion of *generalized* elements: a generalized element of type A is just a map $\Gamma \to A$ (*cf.* a term in context $\Gamma \vdash a : A$). One may systematically construct internal propositions and predicates over generalized elements by using the structure of the subobject classifier Ω, resulting in the so-called *Mitchell-Bénabou language*. We elide the details of the definition of this language — it consists of the usual propositional connectives $\vee, \wedge, \to, \neg$, construed as binary functions $\Omega \times \Omega \to \Omega$, and existential and universal quantification; more details can be found in Mac Lane and Moerdijk [78, Section VI.5].

⚠ **(3.2.2∗3)** It is important to distinguish between internal and external mathematics. For instance, by **(3.2.1.1∗3)**, we observe that the statement “there are three propositions.” is externally valid, the corresponding internal proposition is not, since Ω from an internal perspective simply classifies subsets, and by extension, subsingletons — there is no additional structure aside from those used to construct logical formulas.

(3.2.2∗4) The connection between the internal and external is elucidated via the *Kripke-Joyal semantics* of presheaf categories [78, p. VI.7], which is a series of theorems that unfolds a statement in the internal language of a presheaf category into a statement in external mathematics. By definition, we say that a predicate $\phi : A \to \Omega$ on a generalized element $\Gamma \to A$ holds when the latter factors through the subobject $\{\phi\} \rightarrowtail A$ determined by ϕ, as depicted below:

$$\begin{tikzcd}
 & \{\phi\} \arrow[d, tail] \\
\Gamma \arrow[r] \arrow[ru, dotted] & A
\end{tikzcd}$$

By **(3.1.4∗3)** we have that every Γ is canonically a colimit of representables, so in fact it suffices to consider the cases $\Gamma = \mathsf{y}C$. Since the dotted map is necessarily unique, we write $C \Vdash \phi$ for the relation that holds when such a map exists; we sometimes also call $C \Vdash \phi$ the *forcing relation* and say that C *forces* ϕ.

(3.2.2∗5) The Kripke-Joyal semantics systematically explains the meaning of $C \Vdash \phi$ by structural recursion on formulas ϕ. Here we just give the rules for the case where the base category is a poset, which corresponds to the Kripke semantics of intuitionistic logic [69]. We may unravel the meaning of a predicate $\phi(\alpha)$ over a generalized element $\alpha : \mathsf{y}C \to A$ by recursion on ϕ:

$C \Vdash \top$ always.
$C \Vdash \bot$ never.
$C \Vdash \alpha$ iff $\alpha : \mathsf{y}C \to \Omega$ factors through $\top : 1 \to \Omega$.

$C \Vdash \alpha =_A \beta$ iff $\alpha = \beta : \mathsf{y}C \to A$.
$C \Vdash \phi(\alpha) \vee \psi(\alpha)$ iff $C \Vdash \phi(\alpha)$ or $C \Vdash \psi(\alpha)$.
$C \Vdash \phi(\alpha) \wedge \psi(\alpha)$ iff $C \Vdash \phi(\alpha)$ and $C \Vdash \psi(\alpha)$.
$C \Vdash \phi(\alpha) \to \psi(\alpha)$ iff for all $D \sqsubseteq C$, $D \Vdash \phi(\alpha D)$ implies $D \Vdash \psi(\alpha D)$.
$C \Vdash \exists[b : B]\ \phi(\alpha, b)$ iff there exists $\beta : \mathsf{y}C \to B$ such that $C \Vdash \phi(\alpha, \beta)$.
$C \Vdash \forall[b : B]\ \phi(\alpha, b)$ iff for all $D \sqsubseteq C$ and $\beta : \mathsf{y}D \to B$, $D \Vdash \phi(\alpha D, \beta)$.

In the above, given $C \sqsubseteq D$ and $\alpha : \mathsf{y}C \to A$, we write αD for the generalized element $\mathsf{y}D \to \mathsf{y}C \to A$.

(3.2.2∗6) The forcing semantics $C \Vdash \phi$ is *monotone* in the sense that if $C \Vdash \phi(\alpha)$ and $D \sqsubseteq C$, then $D \Vdash \phi(\alpha D)$. From an epistemic perspective, this means that knowledge is monotone with respect to passage to future worlds.

(3.2.2∗7) As a simple application of the Kripke-Joyal semantics, we show that the *Law of excluded middle* (LEM) is not intuitionistically valid, *i.e.* not a theorem in the internal language of presheaf categories. To do so, we exhibit a countermodel for the statement $\forall[\phi : \Omega]\ (\phi \vee (\phi \to \bot))$ given by the presheaf category $\widehat{\mathbb{I}}$.

(3.2.2∗8) Suppose that LEM holds universally, *i.e.* we have that $C \Vdash \forall[\phi : \Omega]\ \phi \vee (\phi \to \bot)$ for all $C \in \mathbb{I}$. Unfolding the forcing semantics, this means that for all $D \sqsubseteq C$ and $\alpha \in \Omega(D)$, we have that $D \Vdash \alpha$ or $D \Vdash \alpha \to \bot$. To derive a contradiction, it suffices to show that neither $1 \Vdash I$ nor $1 \Vdash I \to \bot$, where $I \in \Omega(1)$ is the intermediate truth value **(3.2.1.1∗1)**.

For the former, by the forcing semantics, we need to show that $I : \mathsf{y}1 \to \Omega$ factors as follows.

$$\begin{tikzcd} \mathsf{y}1 \arrow[r, "I"] \arrow[d] & \Omega \\ 1 \arrow[ur, "\top"'] & \end{tikzcd}$$

In other words, we need to show that the restriction of I along $C \sqsubseteq 1$ for all C is the total sieve t_C on C. But this is not true since I itself is not a total sieve.

For the latter, $1 \Vdash I \to \bot$ unfolds to the statement that for no $C \sqsubseteq 1$ is it the case that $C \Vdash IC$. But this is a contradiction because we have that $0 \sqsubseteq 1$ in $\mathbb{I}$ and $0 \Vdash I0$: since the restriction of I along $0 \sqsubseteq 1$ is the total sieve on 0, we have that $I0 : \mathsf{y}0 \to \Omega$ factors through $\top : 1 \to \Omega$, which, by definition, means that $0 \Vdash I0$ holds.

(3.2.2∗9) This example can also be understood in terms of the semantics of intuitionistic logic in topological spaces in which propositions are valued in the frame[2] of open sets of a space $\mathcal{O}(X)$, and a proposition is "true" when its interpretation is the entire space. In addition to being a frame, the open sets of a space also form a *Heyting Algebra*, *i.e.* a Cartesian closed poset. The exponential of opens $U \to V$ is defined as $\{x \in X \mid x \in U \to x \in V\}^\circ$, where S° is the interior of a subset $S \subseteq X$.

The relevant space in this case is given by the *specialization/upset/Alexandroff* topology on the poset $\mathbb{I} = \{0 \sqsubseteq 1\}$ in which the opens are given by the upwards-closed subsets of $\mathbb{I}$. In other words,

[2]A frame is a lattice with all small joins that satisfies distributivity of meets over joins.

we consider a model in which truth values are open sets of the *Sierpińksi* space $\Sigma = \{\emptyset, \{1\}, \mathbb{I}\}$. The reason LEM fails in this model can be traced to the asymmetric nature of the topology on $\mathbb{I}$: propositions cannot be valued in the subset $\{0\}$, from which a simple calculation shows that $I \vee (I \to \bot)$ holds only on $\{1\}$ and not the total space $\mathbb{I}$.

(3.2.2∗10) In Section 3.4 the connection between the presheaf model and the topological model in the preceding discussions is evinced by the equivalence $\widehat{\mathbb{I}} \cong \mathrm{Sh}(\Sigma)$ where $\mathrm{Sh}(X)$ is the category of *sheaves* on a space X.

3.2.3. Presheaves of algebras.

(3.2.3∗1) As long as the intuitionistic strictures of the internal language are observed, one can carry out a great deal of mathematics internal to presheaf categories. We already explained in **(3.1.3∗3)** that there is a natural numbers object (nno) $\mathbb{N}$ in every presheaf category, and one may work with $\mathbb{N}$ and develop properties of natural numbers in the internal language just as in external mathematics.

(3.2.3∗2) The nno of presheaf categories is given by the constant presheaf on the nno $\mathbb{N}$ in **Set**. One may also ask what are the external meanings of internal mathematical structures that are not necessarily defined by a universal construction. It turns out there is a pleasing explanation for a large class of structures, including all algebraic theories. Given a presheaf category $\widehat{\mathscr{C}}$ and a theory $\mathbb{T}$, a $\mathbb{T}$-algebra internal to $\widehat{\mathscr{C}}$ or an *internal* $\mathbb{T}$*-object* is a $\mathbb{T}$-algebra in the internal language of $\mathbb{T}$. For instance, an internal preorder-object is a presheaf P equipped with a monomorphism $\sqsubseteq : P \times P \to \Omega$ such that the following statements hold in the internal language of $\widehat{\mathscr{C}}$:

1. $\forall[x : P]\ x \sqsubseteq x$.
2. $\forall[x, y, z : P]\ x \sqsubseteq y \to y \sqsubseteq z \to x \sqsubseteq z$.

In conjunction with **(3.1.2∗5)**, one can use the Kripke-Joyal forcing semantics to compute that externally one obtains a preordered set $P(C)$ for every $C : \mathscr{C}$. Moreover, because $\sqsubseteq : P \times P \to \Omega$ is a natural transformation, we have the following diagram for every map $f : C \to D$ in $\mathscr{C}$:

$$\begin{tikzcd}
P^2(D) = (P(D))^2 \arrow[r, "\sqsubseteq_D"] \arrow[d, "P^2(f)"'] & P(D) \arrow[d, "P(f)"] \\
P^2(C) = (P(C))^2 \arrow[r, "\sqsubseteq_C"'] & P(C)
\end{tikzcd}$$

Thus every restriction map $P(D) \to P(C)$ is also a morphism of preorders (*i.e.* a monotone map).

(3.2.3∗3) The general correspondence is that given some algebraic theory $\mathbb{T}$, a $\mathbb{T}$-object internal to a presheaf category $\widehat{\mathscr{C}}$ is just a presheaf of $\mathbb{T}$-algebras in **Set**, *i.e.* we have the following equivalence:

$$\mathbf{Alg}(\mathbb{T}, \widehat{\mathscr{C}}) \simeq [\mathscr{C}^{\mathrm{op}}, \mathbf{Alg}(\mathbb{T}, \mathbf{Set})]$$

Where we write $\mathbf{Alg}(\mathbb{T}, \mathscr{C})$ for the category of $\mathbb{T}$-algebras in $\mathscr{C}$ and homomorphisms.

(3.2.3∗4) Actually the above holds for all *geometric* theories [64, p. D1.2.14], but this generalization is not so interesting for our purposes since the objects we will be dealing with in Chapter 7 are internal *domains*, and these structures are not algebras for geometric theories.

3.2.4. Monads and algebras.

(3.2.4∗1) In Chapters 4 and 7 we will use algebras (not necessarily of algebraic theories) formulated in the internal language of a (pre)sheaf category as a semantic basis for defining and reasoning about programs. In this section we outline a generalization from algebraic theories and their algebras to monads and *monad algebras* (also known as Eilenberg-Moore algebras). In this section we work inside an ambient type theory/internal language of a topos.

(3.2.4∗2) Given a monad $\mathbb{T} = (\mathsf{T}, \eta, \mu)$, a $\mathbb{T}$-*algebra* is a set A equipped with a map $\alpha : \mathsf{T}A \to A$ satisfying the following coherence conditions.

$$\begin{tikzcd} A \arrow[r, "\eta_A"] \arrow[dr, equal] & \mathsf{T}A \arrow[d, "\alpha"] \\ & A \end{tikzcd} \qquad \begin{tikzcd} \mathsf{T}^2A \arrow[r, "\mathsf{T}\alpha"] \arrow[d, "\mu_A"'] & \mathsf{T}A \arrow[d, "\alpha"] \\ \mathsf{T}A \arrow[r, "\alpha"'] & X \end{tikzcd}$$

(3.2.4∗3) Thinking of $\mathsf{T}A$ as an *effectful* computation of A (for instance a cost effect, as we will see in Section 4.1.3) a $\mathbb{T}$-algebra is a type A that comes with an operation $\mathsf{T}A \to A$ that "absorbs"/implements the effect. The coherence laws then state that the provided operation commutes with the existing monad structure in an expected way.

(3.2.4∗4) A $\mathbb{T}$-algebra morphism $X \to Y$ is a function $f : |X| \to |Y|$ of the underlying sets satisfying the following:

$$\begin{tikzcd} \mathsf{T}|X| \arrow[r, "\mathsf{T}f"] \arrow[d] & \mathsf{T}|Y| \arrow[d] \\ |X| \arrow[r, "f"'] & |Y| \end{tikzcd}$$

(3.2.4∗5) The collection of $\mathbb{T}$-algebras and $\mathbb{T}$-algebra morphisms organize into a category $\mathsf{Alg}(\mathbb{T})$ often referred to as the *Eilenberg-Moore category* of $\mathbb{T}$.

(3.2.4∗6) The *free* $\mathbb{T}$-*algebra* on a set A is defined to be $\mathsf{free}_{\mathbb{T}}(A) = (\mathsf{T}A, \mu_A)$. This construction extends to the *free-forgetful adjunction* $\mathbb{T}\text{-}\mathsf{Alg} \underset{\longrightarrow}{\overset{\longleftarrow}{\perp}} \mathbf{Set}$ between the category of $\mathbb{T}$-algebras and the category of plain sets in which the left adjoint sends A to the free $\mathbb{T}$-algebra on A and the right adjoint sends a $\mathbb{T}$-algebra X to the carrier set $|X|$.

(3.2.4∗7) The hom-set bijection $\mathrm{Hom}_{\mathsf{Alg}(\mathbb{T})}(\mathsf{free}_{\mathbb{T}}(A), X) \cong \mathrm{Hom}_{\mathbf{Set}}(A, |X|)$ may be characterized as follows. In the forward direction, an algebra morphism $h : \mathsf{free}_{\mathbb{T}}(A) \to X$ is sent to the composite $A \xrightarrow{\eta_A} \mathsf{T}A \xrightarrow{f} |X|$. In the backward direction, a function $f : A \to |X|$ is sent to $\mathsf{T}A \xrightarrow{Tf} T|X| \xrightarrow{\alpha_X} |X|$.

3.3. THE PHASE DISTINCTION

(3.3∗1) The discussion in Section 3.2.1.1 about propositions in the presheaf category over $\mathbb{I}$ inspires an *internalization* of the intermediate proposition $\mathsf{u} : \Omega$ that will serve as the linchpin for mediating the interaction between the cost-sensitive and functional world as discussed in Section 3.1.1. Ultimately the idea is to organize cost-sensitive data by means of the *open* and *closed modalities* [102] associated to the proposition u, but we will define these modalities relative to an arbitrary proposition in this section.

(3.3∗2) We will use the terms "modality" and "modal" to describe a type-theoretic formulation of cost-sensitive and functional verification (to come in Chapter 4), but a modality in this dissertation interacts with contexts of type theory in a completely standard way (since they are defined in terms of existing constructions, namely function types and (a kind of) quotient types), and thus should be distinguished from "proper" modal type theories in which there exist constructs that are *not* stable under all substitutions; Gratzer [42] is a recent but excellent reference on such modalities in type theory.

(3.3∗3) The *open modality* associated to a proposition ϕ is the function space monad/read monad $\phi \to -$ on ϕ. We also call the open modality the *restriction modality.*

(3.3∗4) The *closed modality* associated to a proposition ϕ is defined as the following quotient inductive type (QIT) [37]:

inductive $\mathsf{u} \vee A : \mathsf{Set}$ **where**

$\eta_{\phi\vee -} : A \to \phi \vee A$
$*_A : \phi \to \phi \vee A$
$\_ : (a : A) \to (u : \phi) \to \eta_{\phi\vee -}(a) = *(u)$

This definition can be understood as an inductive type equipped with two constructors $\eta_{\phi\vee -}$ and $*_A$ that are equated for all $u : \phi$ and $a : A$: $\eta_{\phi\vee -}(a) = *_A(u)$. It is also equivalently defined as the following pushout:

$$\begin{tikzcd}
A \times \phi \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] & \phi \arrow[d, "*_A"] \\
A \arrow[r, "\eta_{\phi\vee -}"'] & \phi \vee A \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

When it is clear from the context we will often omit the type subscript to the constructor $*_A$. We also call the closed modality the *sealing modality.*

(3.3∗5) Both the restriction and sealing modalities are monads; the monadic structure associated to the restriction modality is just the usual *reader monad*, while the monadic structure of the sealing modality is defined as follows (observe that $\eta_{\phi\vee -}$ used in the position of arguments and on the right hand side refer to the *constructor* to the QIT of the sealing modality):

$$\begin{aligned}
&\eta_{\phi\vee -} : A \to \phi \vee A\\
&\eta_{\phi\vee -} = \eta_{\phi\vee -}
\end{aligned}$$

$$\begin{aligned}
&\mu_{\phi\vee -} : \phi \vee (\phi \vee A) \to \phi \vee A\\
&\mu_{\phi\vee -}(*_{\phi\vee A}(p)) = *_A(p)\\
&\mu_{\phi\vee -}(\eta_{\phi\vee -}(*_A(p))) = *_A(p)\\
&\mu_{\phi\vee -}(\eta_{\phi\vee -}(\eta_{\phi\vee -}(a))) = \eta_{\phi\vee -}(a)
\end{aligned}$$

☐ **(3.3∗6)** Both the restriction and sealing modality interact well with many type constructors. A monad $\mathbb{M} = (\mathsf{M}, \eta_{\mathbb{M}}, \mu_{\mathbb{M}})$ is *idempotent* when the unit map $\eta_{\mathbb{M}} : A \to \mathsf{M}A$ is an isomorphism, *lex* when $\mathsf{M}(A \times B) \cong \mathsf{M}A \times \mathsf{M}B$ and $\mathsf{M}(a =_A b) \cong (\eta_{\mathbb{M}}(a) =_{\mathsf{M}A} \eta_{\mathbb{M}}(b))$, and *commutes with exponentials* when $\mathsf{M}(A \to B) \cong (\mathsf{M}A \to \mathsf{M}B)$. We have that the restriction and sealing modality satisfy the following properties.

1. Both the restriction and sealing modality are lex idempotent monads.

2. The restriction modality moreover commutes with exponentials.

📖 **(3.3∗7)** Given an idempotent modality M, we say a type A is *M-modal* when $M(A) \cong A$ and *M-connected* when $M(A) \cong 1$. Observe that $M(A)$ is always M-modal.

(3.3∗8) By the characterization of exponentials in presheaf categories Eq. (3.1.3∗4∗1) we see that in $\widehat{\mathbb{I}}$ the restriction modality sends a family $\begin{array}{c}X(1)\\ \downarrow\\ X(0)\end{array}$ to the family $\begin{array}{c}X(0)\\ \downarrow\\ X(0)\end{array}$ determined by the identity function. A similar calculation shows that the closed modality sends X to the family $\begin{array}{c}X(1)\\ \downarrow\\ 1\end{array}$ determined by the unique map into 1. Thus we have the following description of M-modal and M-connected types in $\widehat{\mathbb{I}}$:

	modal	connected
restriction	$X(1) \cong X(0)$	$X(0) \cong 1$
sealing	$X(0) \cong 1$	$X(1) \cong 1$

In particular, we observe that a sealing-modal (which we may as well shorten to *sealed*) type is always restriction-connected.

💡 **(3.3∗9)** Combining Section 3.1.1 and **(3.3∗8)**, we observe that from an internal perspective, a cost-sensitive computation of type A corresponds exactly to the product type $\mathbb{C} \times A$ for a *sealed* cost monoid $\mathbb{C}$, *i.e.* a type $\mathbb{C}$ such that $\mathbb{C} \cong (\phi \vee \mathbb{C})$.

(3.3∗10) The benefit of this modal decomposition is that one may now describe a cost-sensitive function (in the language of presheaves over $\mathbb{I}$) in terms of a *single* function instead of a square in $\widehat{\mathbb{I}}$.

(3.3∗11) Another crucial benefit is the ability to *strip away* cost structure when we need to reason about programs in a purely functional capacity, *i.e.* correctness specifications. This is achieved by means of the restriction modality. By **(3.3∗8)**, we have that any sealed type is trivial in the presence of the proposition u, which in particular includes the type of costs $\mathbb{C}$. Thinking about cost structure as encoded in the total space of a family X, the restriction modality can be understood literally as the restriction action of X as a presheaf: one is restricted to the world at the base in which cost structure is trivialized.

(3.3∗12) We may call any context in which one has assumed the proposition u as the *functional phase*, in contradistinction to the cost-sensitive phase. By definition, we have that cost structure is trivial in the functional phase.

(3.3∗13) The observation about noninterference of cost and functional semantics in **(3.1.1∗3)** can also be phrased in terms of the restriction and sealing modalities: *every function $A \to B$ from a sealed type to a restriction-modal type is constant.* In other words, one cannot branch on cost information to produce distinct functional semantics.

■ **(3.3∗14)** The proof of **(3.3∗13)** follows from purely formal properties of the open and closed modalities associated to a proposition. By definition of modal types, every $f : A \to B$ determines a function $(\phi \vee A) \to (\phi \to B)$; by a tranpose of arguments, this is equivalent to a function $\phi \to (\phi \vee A) \to B$, which is determined by a single point $b : B$ since $\phi \vee A$ is sealed.

(3.3∗15) In Chapter 4 we show how these observations give rise to a modal[3] type theory for doing cost-sensitive programming and verification.

3.4. SHEAVES

(3.4∗1) The Yoneda embedding presents presheaf categories as the *free cocompletion* of their base categories, in the sense that every functor $\mathscr{C} \to \mathscr{E}$ into a cocomplete category $\mathscr{E}$ extends to an essentially unique colimit-preserving functor $\widehat{\mathscr{C}} \to \mathscr{E}$, as shown below:

$$\begin{tikzcd} \mathscr{C} \arrow[r, "\mathsf{y}"] \arrow[d] & \widehat{\mathscr{C}} \arrow[dl, dotted] \\ \mathscr{E} & \end{tikzcd} \tag{3.4∗1∗1}$$

(3.4∗2) Similar to the situation of the free monoid or a free group on a set, the free cocompletion of a category can be thought of as adjoining to the base category $\mathscr{C}$ all *formal* colimits. Consequently, this means that existing colimits of $\mathscr{C}$ need *not* be preserved by the embedding $\mathsf{y}_{\mathscr{C}} : \mathscr{C} \to \widehat{\mathscr{C}}$. For instance, the initial object 0 of interval category $\mathbb{I}$ is sent to $\mathsf{y}_{\mathbb{I}}(0)$, which is the intermediate proposition u in $\widehat{\mathscr{C}}$ and not the initial object $0_{\widehat{\mathscr{C}}}$ (which is defined as the constant presheaf valued in $0_{\mathbf{Set}}$).

(3.4∗3) The failure to preserve colimits will pose an obstacle to us in Chapter 7, where we will use the Yoneda embedding to construct models of *synthetic domain theory*, which provides a way to integrate general recursion in dependent type theory. The short of it is that we would like the

[3]As discussed in **(3.3∗2)**, the type theories we work with in this dissertation are all structural, *i.e.* the rules governing the modalities do not restrict or modify the context in any way.

Yoneda embedding to preserve finite coproducts (so in particular the initial object), and the way to accomplish this is by means of *sheaves*.

(3.4∗4) We will mostly understand sheaves as a way to control the behavior of colimits under the Yoneda embedding. For an introduction to the (truly vast) theory and applications of sheaves (especially pertaining to logic and type theory) we refer the reader to the standard reference [78].

3.4.1. Sheaves on spaces.

(3.4.1∗1) In this section we fix a category $\mathscr{C}$ with finite limits.

(3.4.1∗2) A *sheaf* on $\mathscr{C}$ is a just presheaf on $\mathscr{C}$ satisfying some (seemingly) bizarre conditions, which are organized into a structure called a *site* consisting of the base category $\mathscr{C}$ and a *coverage* K (though in prose we will often just refer to the base category as the site when the coverage is evident). The category of sheaves on a site $(\mathscr{C}, K)$ is then a full subcategory of the category of presheaves on $\mathscr{C}$.

How do sheaves (more precisely, the coverage) control the behavior of colimits? Later in this section I will outline the technical details of how this is achieved, but at this point it may help to have some rough pictures in mind. To be specific, we are interested in preserving colimits that are stable under pullbacks in the sense that when $\{C_i \to C\}_i$ is a colimit diagram then we may pull back along any $f : D \to C$ to obtain another colimit diagram $\{f^*C_i \to D\}_i$.[4] The role of the coverage is to isolate a class of pullback-stable diagrams that are designated to be sent to colimits in the resulting sheaf category, which is ensured by the sheaf condition.

For simplicity, suppose that the base category $\mathscr{C}$ has a class of well-behaved colimits J in the sense just described. Roughly, a coverage K is an assignment of C to a set of covers of C (a set of morphisms $\{C_i \to C\}_i$ with codomain C). The preservation of colimits $\{C_i \to C\}_i \in J$ is specified by the coverage K in which $K(C)$ includes $\{C_i \to C\}_i$ when C is a colimit in J (but is otherwise free to include/exclude other covers, subject to pullback stability). The sheaf condition relative to K then ensures that every sheaf "thinks" the image of a specified colimit $\{C_i \to C\}_i$ under the Yoneda embedding is a colimit diagram; by *think* I mean that $\{\mathrm{y}C_i \to \mathrm{y}C\}_i$ has the universal mapping property for all *sheaves*. Thus a coverage is essentially a declaration of the class of colimits to be preserved, and the sheaf condition is just the property needed to execute this declaration.

(3.4.1∗3) Note that in general a coverage may be used to isolate certain pullback-stable families in the base category that are *not* necessarily colimit diagrams. Thus one can say that a coverage sends diagrams that *look* like they should be colimits to actual colimits.

(3.4.1∗4) A *sink* on an object $C : \mathscr{C}$ is a collection of arrows into C. Given a sink $\underline{C} = \{C_i \to C\}_i$ and $f : D \to C$, we write $f^*\underline{C}$ for the sink $\{C_i \times_C D \to D\}_i$ defined by pulling back each $C_i \to C$ along f.

(3.4.1∗5) A (Cartesian) *coverage* is an assignment K of objects C to collections of sinks on C called *covers* that is stable under pullback, in the sense that given $\underline{C} \in K(C)$ and $f : D \to C$ we have $f^*\underline{C} \in K(D)$.

[4]The type-theoretic significance of this requirement is that we want to consider colimits that are well-behaved with respect to substitutions.

(3.4.1*6) Observe that a coverage defines a contravariant functor $\mathscr{C}^{\mathrm{op}} \to \mathbf{Set}$.

(3.4.1*7) Given a presheaf $X : \mathscr{C}^{\mathrm{op}} \to \mathbf{Set}$, a *matching family* for a cover $\underline{C} = \{C_i \to C\}_i$ is an assignment $x_i \in X(C_i)$ of elements to each $C_i \to C$ such that for every pair of generalized elements $f : \Gamma \to C_i$ and $g : \Gamma \to C_j$ configured as below

$$\begin{tikzcd}
\Gamma \arrow[r, "f"] \arrow[d, "g"'] & C_i \arrow[d] \\
C_j \arrow[r] & C
\end{tikzcd}$$

we have that the restriction of x_i along f is equal to the restriction of x_j along g.

(3.4.1*8) A *sheaf* with respect to a coverage K is a presheaf $X : \mathscr{C}^{\mathrm{op}} \to \mathbf{Set}$ such that every for matching family $\{x_i\}_i$ for a cover $\underline{C}$ extends to a unique element $x \in X(C)$, *i.e.* we have that x restricts to x_i along $C_i \to C$.

(3.4.1*9) A *site* $(\mathscr{C}, K)$ is a small category equippped with a coverage K. We write $\mathrm{Sh}(C, K)$ for the full subcategory of $\widehat{\mathscr{C}}$ consisting of K-sheaves.

(3.4.1*10) A *sheaf topos* is a category equivalent to a category of sheaves on a site.

(3.4.1*11) Recall from **(3.2.1*4)** that a sieve on C is a subobject of the representable presheaf $\mathsf{y}C$, *i.e.* a sink on C closed under precomposition. Every sink generates a sieve by precomposition; we write $(\underline{C})$ for the sieve generated by a sink $\underline{C}$.

(3.4.1*12) Observe that every matching family for a cover $\underline{C}$ on a presheaf X extends to a natural transformation $(\underline{C}) \to X$. Since an element in $X(C)$ is in one-to-one correspondence with natural transformations $\mathsf{y}C \to X$, we have that a presheaf X is a sheaf just when for all covers $\underline{C}$, a map $(\underline{C}) \to X$ extends uniquely along $(\underline{C}) \to \mathsf{y}C$:

$$\begin{tikzcd}
(\underline{C}) \arrow[r, tail] \arrow[d] & \mathsf{y}C \arrow[dl, dotted] \\
X &
\end{tikzcd}$$

(3.4.1*13) A coverage is *subcanonical* when every representable presheaf is a sheaf.

(3.4.1*14) A presheaf is a sheaf with respect to the *trivial coverage*, in which a sink $\underline{C}$ covers C if and only if $(\underline{C}) = \mathsf{y}C$, *i.e.* if and only if $\underline{C}$ contains the identity $1 : C \to C$.

(3.4.1*15) The frame of opens of a topological space can be equipped with the *open cover coverage* whose covers are covers in the topological sense, *i.e.* $\{U_i \to U\}_i$ is a cover when U_i are open subspaces of U and $\bigvee_i U_i = U$. We write $\mathrm{Sh}(X)$ for the category of sheaves on $\mathcal{O}(X)$ equipped with the open cover coverage.

(3.4.1*16) Consider the category of sheaves $\mathrm{Sh}(\Sigma)$ on the Sierpiński space Σ, defined as the space of up-sets on the interval $\mathbb{I}$. As claimed in **(3.2.2*10)**, we have that $\mathrm{Sh}(\Sigma) \cong \widehat{\mathbb{I}}$.

■ **(3.4.1∗17)** Because the empty cover $\emptyset$ is an open cover of the open $\emptyset$, we have that a sheaf X in $\mathrm{Sh}(\Sigma)$ satisfies the following unique extension property:

$$\begin{tikzcd}
\emptyset \arrow[r, tail] \arrow[d] & \mathsf{y}\emptyset \arrow[dl, dotted, "\alpha"] \\
X &
\end{tikzcd} \tag{3.4.1∗17∗1}$$

Since $\emptyset \to \emptyset$ is the only map into the empty set, we have that $\alpha : \mathsf{y}\emptyset \to X$ is determined by a single point $x \in X(\emptyset)$, which must be unique by Eq. (3.4.1∗17∗1). This means that $X(\emptyset) \cong 1$. Thus the data of a sheaf on Σ is completely determined by the poset $\{\{1\} \sqsubseteq \Sigma\} \cong \mathbb{I}$. In other words, a sheaf on Σ is equivalent to a presheaf on $\mathbb{I}$.

3.4.2. Sheaves as conservative cocompletions.

(3.4.2∗1) In light of **(3.4∗2)**, sheaves become relevant as a way to specify *conservative* cocompletions.

(3.4.2∗2) A *diagram scheme* is a small category I serving as the indexing category of a diagram.

(3.4.2∗3) Given a set of diagram schemes Φ and categories $\mathscr{C}$, $\mathscr{D}$ with all Φ-colimits, a functor $\mathscr{C} \to \mathscr{D}$ is Φ*-cocontinuous* when it preserves all Φ-colimits.

(3.4.2∗4) Given a set of diagram schemes Φ and a category $\mathscr{C}$, an Φ*-conservative cocompletion* is a Φ-cocontinuous functor $\mathscr{C} \to \tilde{\mathscr{C}}$ into a cocomplete category $\tilde{\mathscr{C}}$ satisfying the following unique extension property for every Φ-cocontinuous $\mathscr{C} \to \mathscr{E}$:

$$\begin{tikzcd}
\mathscr{C} \arrow[r, tail] \arrow[d] & \tilde{\mathscr{C}} \arrow[dl, dotted] \\
\mathscr{E} &
\end{tikzcd}$$

☐ **(3.4.2∗5)** For every set of diagram schemes Φ, the Yoneda embedding $\mathscr{C} \to \widehat{\mathscr{C}}$ restricts to a Φ-conservative cocompletion $\mathscr{C} \to \tilde{\mathscr{C}}$ [67].

(3.4.2∗6) To see the connection to sheaves, let us derive the necessary consequences of $\mathsf{y} : \mathscr{C} \to \widehat{C}$ restricting to a Φ-conservative cocompletion. Let $D : I \to \mathscr{C}$ be a Φ-diagram, and write $\dot{C}$ for a cocone whose vertex is C. We must have that any colimiting cocone $\dot{C} \in \mathsf{Cocone}(D)$ is sent to a colimiting cocone $\mathsf{y}\dot{C} \in \mathsf{Cone}(\mathsf{y}D)$. By the universal property of colimits, this means that for every cocone $X \in \mathsf{Cocone}(\mathsf{y}D)$, there is a unique natural transformation $\mathsf{y}C \to X$. Viewing a cocone with vertex C as a sink on C, this is equivalent to the following unique extension property for every presheaf X:

$$\begin{tikzcd}(\dot{C}) \arrow[r, tail] \arrow[d] & \mathsf{y}C \arrow[dl, dotted] \\ X & \end{tikzcd}$$

By **(3.4.1∗12)**, this is just saying that X is sheaf with respect to the coverage in which every colimiting cocone is a cover.

(3.4.2∗7) More explicitly, we may observe that a cocone $X \in \mathsf{Cocone}(\mathsf{y}D)$ corresponds to a family $\{\mathsf{y}D_i \to X(i)\}_{i\in I}$ such that for all $i \to j \in I$, we have that the following diagram commutes:

$$\begin{tikzcd}\mathsf{y}D_i \arrow[r] \arrow[d] & \mathsf{y}D_j \arrow[dl] \\ X & \end{tikzcd}$$

By the Yoneda lemma, this is equivalent to a family $\{x_i \in X(D_i)\}_{i\in I}$ such that x_i is the restriction of x_j for every $i \to j$, which is just the *limit* of the diagram $XD^{\mathrm{op}} : I^{\mathrm{op}} \to \mathbf{Set}$. Moreover, we observed in **(3.4.2∗6)** that cocones $X \in \mathsf{Cocone}(\mathsf{y}D)$ are in bijective correspondence with elements of $X(C)$, where C is the colimiting cocone of D. In other words, we have that a presheaf in the Φ-conservative cocompletion $\tilde{\mathscr{C}}$ must send Φ-colimits in $\mathscr{C}$ to Φ^{op}-limits in **Set**.

(3.4.2∗8) By **(3.4.2∗5)**, every coverage arising from a conservative cocompletion is subcanonical.

(3.4.2∗9) In general conservative cocompletions need not give rise to a sheaf topos; for instance there is no way to obtain a coverage when the colimit in the base category is not pullback stable (as is the case for the joins of a poset, which are not even disjoint!) On the other hand, not every category of sheaves on a site necessarily comes from a conservative cocompletion either, since by **(3.4.2∗8)** we know that the latter (when it comes from a coverage) must be subcanonical.

(3.4.2∗10) By **(3.4.1∗17)**, we have that a sheaf X on the Sierpiński space sends the empty open to the one-point set, so we may view the embedding $\mathsf{y} : \mathcal{O}(\Sigma) \to \mathrm{Sh}(\Sigma)$ as the initial-object-conservative cocompletion of $\mathcal{O}(\Sigma)$.

(3.4.2∗11) Most interesting cases of conservative cocompletion come not from "petit" toposes coming from a single space, but from "gros" toposes whose sites are thought of as *categories* of spaces. For instance, in Chapter 7 we consider a site of *domains* equipped with a "gros" version of the open cover topology called the *extensive topology.*

(3.4.2∗12) A category with finite coproducts is *extensive* when the following the canonical functor $\mathscr{C}/A \times \mathscr{C}/B \to \mathscr{C}/(A + B)$ sending $C \to A, D \to B$ to $C + D \to A + B$ is an equivalence.

(3.4.2∗13) Intuitive, extensivity says that sums are well-behaved. For instance, every extensive category with finite products, products distribute over sums.

(3.4.2∗14) Finite coproducts are extensive just when coproduct injections are stable under pullbacks. This means that given a pullback situation as follows.

$$\begin{tikzcd}
Y \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d] \\
X \arrow[r] & A+B \\
Z \arrow[r] \arrow[u] \arrow[ur, phantom, "\urcorner", very near start] & B \arrow[u]
\end{tikzcd}$$

we have that $Y \to X \leftarrow Z$ is a coproduct diagram.

(3.4.2∗15) Observe that by **(3.4.2∗14)**, every small extensive category $\mathscr{C}$ is equipped with a coverage defined by all coproduct injections. We call this the *extensive coverage.*

(3.4.2∗16) By **(3.4.2∗5)** and **(3.4.2∗6)**, we have that the restricted Yoneda embedding $\mathscr{C} \to \mathrm{Sh}(\mathscr{C})$ on the extensive site on $\mathscr{C}$ preserves finite coproducts.

3.5. LOCALIZATIONS AND REFLECTIVE SUBUNIVERSES

(3.5∗1) Much of the technical development in Chapter 7 about synthetic domain theory is best understood in terms of (internal) *localizations*, which in turn generate *reflective subuniverses* of (pre)domains that smooth integrates partiality into dependent type theory. In this section we define and build some intuitions about these notions.

(3.5∗2) An object X is *orthogonal* to a morphism $f : A \to B$ when the following unique extension property holds:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d] & B \arrow[dl, dotted] \\
X &
\end{tikzcd}$$

In other words, we have that the precomposition map $X^f : \mathrm{Hom}(B,X) \to \mathrm{Hom}(A,X)$ is an isomorphism.

(3.5∗3) In a Cartesian closed category, an object X is *internally orthogonal* to a morphism $f : A \to B$ when it is orthogonal to every $Z \times f : Z \times A \to Z \times B$.

(3.5∗4) In a topos $\mathscr{E}$, an object X is internally orthogonal to $f : A \to B$ when the unique extension property holds in an internal sense, *i.e.* when the following is true in the internal language of $\mathscr{E}$:

$$\forall[g : A \to X]\ \exists![\overline{g} : B \to X]\ g = \overline{g}f$$

✎ **(3.5∗5)** A proposition $\phi : \Omega$ can be equivalently defined as an object that is internally orthogonal to the unique map $2 \to 1$.

✎ **(3.5∗6)** Recalling the idea of M-modal types from Section 3.3, we have that a type is restriction-modal if and only if it is internally orthogonal to $\mathsf{u} \to 1$ and sealing-modal or sealed if and only if it is internally orthogonal to $0 \to \mathsf{u}$. Intuitively, we might say that a restriction-modal type "thinks" u is true and a sealed type thinks u is false.

(3.5∗7) Given X (internally) orthogonal to f, we also say that it is *(internally) f-local.*

🕮 **(3.5∗8)** Given a set of maps S in a category $\mathscr{C}$, the *(internal) localization* of $\mathscr{C}$ at S is defined as the full subcategory of (internal) S-local objects.

✎ **(3.5∗9)** Observe that by **(3.4.1∗12)**, the category of sheaves $\mathrm{Sh}(\mathscr{C}, K)$ on a site K is just the localization of $\widehat{\mathscr{C}}$ at the collection of sieves induced by the coverage K.

3.5.1. Reflective subcategories.

🕮 **(3.5.1∗1)** A *reflective subcategory* of a category $\mathscr{C}$ is a full subcategory $\mathscr{D} \hookrightarrow \mathscr{C}$ equipped with a left adjoint $L : \mathscr{C} \to \mathscr{D}$ to the inclusion functor called the *reflector.*

🕮 **(3.5.1∗2)** An *exponential ideal* of a Cartesian closed category $\mathscr{C}$ is a full subcategory $\mathscr{D} \hookrightarrow \mathscr{C}$ such that for every $D \in \mathscr{D}$ and $C \in \mathscr{C}$, we have that $D^C \in \mathscr{D}$.

☐ **(3.5.1∗3)** Every internal localization of a (pre)sheaf category[5] is a *reflective exponential ideal* [111].

(3.5.1∗4) Reflective exponential ideals $\mathscr{D} \underset{\hookrightarrow}{\overset{\longleftarrow}{\perp}} \mathscr{C}$ are very well-behaved: since the inclusion is a right adjoint (to the reflector), the subcategory $\mathscr{D}$ is closed under limits, and by definition, exponentials. Moreover, $\mathscr{D}$ has all colimits that exist in $\mathscr{C}$ (though they are not usually preserved by the inclusion $\mathscr{D} \hookrightarrow \mathscr{C}$). In particular, this means that every reflective exponential ideal of a (pre)sheaf topos is cocomplete.

🕮 **(3.5.1∗5)** Rijke, Shulman, and Spitters [102, Theorem A.18] shows that every reflective exponential ideal induces a *reflective subfibration* [102, Definition A.3], which is a system of reflective subcategories $\mathscr{D}_X \underset{\hookrightarrow}{\overset{\xleftarrow{L_X}}{\perp}} \mathscr{C}/X$ satisfying the following properties.

1. *Stability under pullback*: the pullback functors $f^* : \mathscr{C}/Y \to \mathscr{C}/X$ restrict to functors $\mathscr{D}_Y \to \mathscr{D}_X$.

2. *Reflectors commute with pullback*: for any $Z : \mathscr{C}/Y$ the map $L_X(f^*Z) \to f^*(L_Y Z)$ induced from the unit $\eta : Z \to L_Y Z$ is an isomorphism.

(3.5.1∗6) In the internal language of a (pre)sheaf topos, a reflective subfibration may be presented as a *reflective subuniverse*, which corresponds to a notion of an *internal* reflective subcategory [109]; the data of this structure can be presented in type-theoretic language as follows (assuming there is a type-theoretic universe $\mathcal{U}$ in the sense of Hofmann and Streicher [60]).

[5]This result extends to more general categories, but (pre)sheaf categories suffice for our purposes.

$X : \mathcal{U} \vdash \mathsf{inRsc}(X) : \Omega$
$X : \mathcal{U} \vdash L(X) : \mathcal{U}$
$X : \mathcal{U} \vdash \mathsf{inRsc}(L(X)) = \top$
$X : \mathcal{U} \vdash \eta_X : X \to L(X)$
$X, Y : \mathcal{U}, u : \mathsf{inRsc}(X) \vdash \forall[g : Y \to X]\ \exists![\overline{g} : L(Y) \to X]\ g = \overline{g}\eta_Y$

In essence a reflective subuniverse is a reflective subcategory that makes sense in every context, which is a crucial property for developing mathematics in an internal style; we shall exploit reflective subuniverses in both Chapter 4 and Chapter 7 to integrate different classes of types (in the sense of different M-modal types) into a single theory. Although we do not rely on this fact, one may also define both the proposition of being in the reflective subcategory inRsc and the reflection of a type X by means of quotient inductive types [110].

(3.5.1∗7) *Sheafification* $\mathrm{Sh}(\mathscr{C}) \underset{\hookrightarrow}{\overset{\longleftarrow}{\perp}} \widehat{\mathscr{C}}$ induces a reflective exponential ideal, and thus $\mathrm{Sh}(C)$ is a reflective subuniverse in $\widehat{\mathscr{C}}$.

(3.5.1∗8) A reflective subuniverse $\mathcal{U}$ of a topos $\mathscr{E}$ is Σ-*closed* when it is closed under the dependent sum types of $\mathscr{E}$.

(3.5.1∗9) Both restriction-modal and sealing-modal types form Σ-closed reflective subuniverses $\mathcal{U}_R$ and $\mathcal{U}_S$ in a topos. One may think of $\mathcal{U}_R$ and $\mathcal{U}_S$ as classifying purely *functional* and purely *cost-sensitive* data, respectively.

CHAPTER 4

A cost-aware logical framework

4.1. RECONCILING COST-SENSITIVE AND FUNCTIONAL SEMANTICS

(4.1∗1) We are now finally equipped to explain the solution to the problem in Section 0.3 in a mathematically rigorous manner. Recall that the challenge is to study programs both in a cost-sensitive manner and *qua* functions in a unified theory. The tension between the two perspectives is resolved by means of the *functional* vs. *cost-sensitive phase distinction* (FC-phase distinction for short) in the sense of Section 3.3.

(4.1∗2) A *model of the FC-phase distinction* is a category $\mathscr{C}$ with a subterminal object ¶.

(4.1∗3) In the following, we work internally to a topos model $(\mathscr{E}, ¶)$ of the FC-phase distinction.

(4.1∗4) For generality, we work axiomatically, but for intuition one should think in terms of the canonical topos model of the FC-phase distinction $(\widehat{\mathbb{I}}, \mathsf{u})$.

(4.1∗5) The cost structure of programs is tracked by a *sealed* monoid. In the following we fix such a monoid $(\mathbb{C}, 0, +)$.

(4.1∗6) Given any monoid $(M, 0, +)$, the sealing modality induces a monoid $(\bullet M, \eta_{\bullet -}(0), \bullet +)$; the monoid action is defined using the functorial action of $\bullet -$, and has the following explicit description:

$$
\begin{aligned}
&+_\bullet : \bullet M \times \bullet M \to \bullet M\\
&\eta_{\bullet -}(m) +_\bullet \eta_{\bullet -}(n) = \eta_{\bullet -}(m+n)\\
&*u +_\bullet - = *u\\
&- +_\bullet *u = *u
\end{aligned}
$$

In other words, $\bullet +$ acts accordingly on elements of the underlying monoid M and restricts to the identity map $1 \cong 1 \times 1 \to 1$ in the functional phase. Note that there is a similar lifting of any algebraic structure to a corresponding sealed structure by virtue of the fact that the sealing modality preserves finite products **(3.3∗6)**.

(4.1∗7) Recalling Section 3.2.3, a monoid internal to a presheaf topos is a presheaf of monoids in **Set**. Combined with the characterization in **(3.3∗8)**, a sealed monoid in $\widehat{\mathbb{I}}$ corresponds externally to a family $\begin{array}{c} M \\ \downarrow \\ 1 \end{array}$ in which the total space M is a monoid in **Set**.

(4.1∗8) A *cost-sensitive* function is a function of the form $A \to \mathbb{C} \times B$.

(4.1∗9) One may reason about cost-sensitive functions as usual. For instance, we may define a version of insertion sort that tracks the number of the comparison operations $\mathit{lt} : \mathbb{N} \to \mathbb{N} \to 2$:

$\mathit{insert} : \mathbb{N} \times \mathsf{list} \to \bullet\mathbb{N} \times \mathsf{list}$
$\mathit{insert}(x, \mathsf{nil}) = (\eta_{\bullet-}(0), [x])$
$\mathit{insert}(x, y :: l) = \mathsf{if}(\mathit{lt}(x, y), (\eta_{\bullet-}(1), x :: y :: l), \mathsf{let}\ (c, l') = (\mathit{insert}(x, l))\ \mathsf{in}\ (\eta_{\bullet-}(1)\ (\bullet+)\ c, y :: l'))$

$\mathit{insertSort} : \mathsf{list} \to \bullet\mathbb{N} \times \mathsf{list}$
$\mathit{insertSort}(\mathsf{nil}) = (\eta_{\bullet-}(0), \mathsf{nil})$
$\mathit{insertSort}(x :: l) = \mathsf{let}\ (c, l') = \mathit{insertSort}(l)\ \mathsf{in}\ \mathsf{let}\ (c', l'') = \mathit{insert}(x, l')\ \mathsf{in}\ (c \bullet\!+ c', l'')$

Because we are simply recording a discrete natural number value in this example, we instantiate the cost monoid $\mathbb{C}$ with the concrete sealed monoid $\bullet\mathbb{N}$. Observe that the cost of sequential composition of functions is tracked by the monoid operation $\bullet+$. In a similar vein, we may define[1] a cost-sensitive function $\mathsf{mergeSort} : \mathsf{list} \to \bullet\mathbb{N} \times \mathsf{list}$ that tracks the number of uses of the comparison operator in the merge sort algorithm.

4.1.1. Cost bounds.

(4.1.1∗1) A monoid $(M, 0, +)$ is *left cancellative* when $+ : M \times M \to M$ is injective in the first component.

(4.1.1∗2) To define cost bounds we need a notion of preorder that is compatible with the monoid operation. A *preordered monoid* is a monoid $(M, 0, +)$ equipped with a preorder relation $\sqsubseteq$ such that $+ : M \times M \to M$ is monotone with respect to $\sqsubseteq$.

(4.1.1∗3) Observe that every left cancellative monoid is equipped with the *prefix preorder* in which $x \sqsubseteq y \overset{\text{def}}{=} \Sigma_{k:M}.x + k = y$.[2] As in **(4.1∗6)**, this means the sealing modality also induces a sealed preorder on every monoid:

$\sqsubseteq_\bullet : \bullet M \to \bullet M \to \bullet\Omega$
$\eta_{\bullet-}(m) \sqsubseteq_\bullet \eta_{\bullet-}(n) = \eta_{\bullet-}(m \sqsubseteq n)$
$*u \sqsubseteq_\bullet - = *u$
$- \sqsubseteq_\bullet *u = *u$

Like for the sealed monoid operation, we have that $\sqsubseteq_\bullet$ restricts to the total relation on 1×1 in the functional phase.

(4.1.1∗4) Using the fact that the sealing monad commutes with dependent sums, we may compute that $\bullet(m \sqsubseteq n)$ holds if and only if $\Sigma_{k^\bullet : \bullet M}.k^\bullet + \eta_{\bullet-}(n) = \eta_{\bullet-}(m)$, which by definition is $\eta_\bullet(m) \sqsubseteq_\bullet \eta_\bullet(n)$.

[1]This is not immediately straightforward at the moment because the natural way to write merge sort requires non-structural recursion. We will explain how to deal with such functions in Section 4.2.3.

[2]Technically every monoid M is equipped with a prefix relation, but $x \sqsubseteq y$ is not subterminal unless M is left cancellative.

✎ **(4.1.1∗5)** The prefix preorder on $\mathbb{N}$ equipped with its usual monoid structure is the usual preorder on $\mathbb{N}$.

📖 **(4.1.1∗6)** Given a sealed preordered monoid $\mathbb{C}$, a *cost (upper) bound* of a cost-sensitive function $f : A \to \mathbb{C} \times B$ is a function $\Phi : A \to \mathbb{C}$ satisfying the proposition $\forall[a : A]\ (f(a) \cdot 1) \sqsubseteq \Phi(a)$.

✎ **(4.1.1∗7)** One may establish cost bounds on cost-sensitive functions by ordinary (in)equational reasoning. For instance, we may check that the function $\eta_{\bullet -}| - |^2 : \mathsf{list} \to \bullet\mathbb{N}$ sending a list to the (sealed) square of its length is a cost bound for *insertSort*.

4.1.2. Functional reasoning.

(4.1.2∗1) More interestingly, the fact that the cost monoid $\mathbb{C}$ is sealed enables us to smoothly transition to pure functional reasoning as well. For instance, the following equation holds in the *functional phase*:

$$\mathit{insertSort} = \mathit{mergeSort} \tag{4.1.2∗1∗1}$$

In other words, we have that $\bigcirc(\mathit{insertSort} = \mathit{mergeSort})$ holds, writing $\bigcirc A$ for the restriction modality $\P \to -$ associated to $\P : \Omega$. The reason is that the functional component of a sealed monoid is trivial, and accordingly the monoid operation restricts to the identity map **(4.1∗6)**. In the canonical model, we may visualize this as insertion sort and merge sort projecting to the same underlying function $\mathsf{list} \to \mathsf{list}$.

(4.1.2∗2) The fact that *insertSort* and *mergeSort* are actually *equal* (and not just equivalent up to some relation) has important consequences for reasoning about programs as *functions*. For instance, every sorting procedure should satisfy the following properties:

1. The output list should be a permutation of the input.
2. The length of the output list should equal the length of the input list (this is implied by the previous property).
3. The output list should be in ascending order.

Because the above properties are entirely about the functional aspect of programs, we may defined them as *functional properties* on functions of the form $\mathsf{list} \to \mathbb{C} \times \mathsf{list}$, *i.e.* predicates $(\mathsf{list} \to \mathbb{C} \times \mathsf{list}) \to \Omega_R$, where Ω_R is the universe of restriction-modal propositions.

(4.1.2∗3) By defining functional properties to be valued in restriction-modal propositions, we may immediately translate functional properties across functional equalities such as Eq. (4.1.2∗1∗1). For instance, letting $\mathsf{isSort} : (\mathsf{list} \to \mathbb{C} \times \mathsf{list}) \to \Omega_R$ be the conjunction of **(4.1.2∗2)**, we have that $\mathsf{isSort}(\mathit{insertSort})$ if and only if $\mathsf{isSort}(\mathit{mergeSort})$ holds: by the definition of a restriction-modal proposition, we have that $\mathsf{isSort}(l) = (\P \to \mathsf{isSort}(l))$, from which the result follows since $\P \to (\mathit{insertSort} = \mathit{mergeSort})$.

4.1.3. Cost as an abstract effect.

(4.1.3∗1) So far we have integrated both functional and cost-sensitive reasoning in a unified language. But there is still a problem: by treating cost as a *concrete* cost effect (in the form of a writer monad), the notion of cost-sensitive functions as defined in **(4.1∗8)** allows for some unexpected behaviors when used as a *programming language*.

(4.1.3∗2) For instance, in the definition of *insertSort*, we could have accidentally neglected to propogate the cumulative cost from the recursive call:

$$\begin{array}{l} \ldots \\ \mathit{insertSort}(x :: l) = \mathsf{let}\ (c, l') = \mathit{insertSort}(l)\ \mathsf{in}\ \mathsf{let}\ (c', l'') = \mathit{insert}(x, l')\ \mathsf{in}\ (c', l'') \end{array}$$

This would enable one to unintentionally derive an unexpectedly low cost bound for insertion sort (with respect to the comparison cost metric).

(4.1.3∗3) To prevent such programming errors, one puts the mechanism of cost aggregation behind an *abstraction* that furnishes only the essential operations (and not accidental properties such as the fact that cost structure is represented by means of tuples).

(4.1.3∗4) There are multiple different abstractions one may choose for the concrete effect of incurring cost. Here we will choose to represent cost as a *call-by-push-value* effect [73], for the reason that they are more natural in dependent type theories (as compared to the somewhat more well-known *monadic effects* [85]).

(4.1.3∗5) The idea behind call-by-push-value is to stratify types into *value types* and *computation types*, representing the fact that one may either classify programs as pure functions or classify programs as cost-effectful/cost-sensitive functions. Consequently, from a semantic point of view, a value type corresponds to a plain set A, and a computation type corresponds to a *cost algebra* X, *i.e.* a set equipped with a structure map $\mathbb{C} \times X \to X$ satisfying some coherence conditions with respect to the monoid structure on $\mathbb{C}$. Intuitively one may think about a cost algebra as a type X that supports a cost effect operation $\mathbb{C} \times X \to X$. For instance, the *free* cost algebra is just the set $\mathbb{C} \times A$ equipped with the structure map $\mathbb{C} \times (\mathbb{C} \times A) \to \mathbb{C} \times A$ sending $(c_1, (c_2, a))$ to $(c_1 + c_2, a)$.

(4.1.3∗6) The type structure of the call-by-push-value language(s) used in this thesis is generated from the adjunction $\mathsf{Alg} \underset{\longrightarrow}{\overset{\longleftarrow}{\perp}} \mathbf{Set}$ in which the left adjoint $\mathsf{F} : \mathbf{Set} \to \mathsf{Alg}$ sends a set A to the free cost algebra on A and the right adjoint $\mathsf{U} : \mathsf{Alg} \to \mathbf{Set}$ sends an algebra X to the underlying set of points $|X|$. Product and function types bifurcate into the following four types:

1. Pure functions $A \to B$.
2. Pure products $A \times B$.
3. Cost-sensitive functions $A \to X$.
4. Cost-sensitive products $X \times Y$.

Thus in addition to ordinary products and functions, in a call-by-push-value language cost structure may be manipulated compositionally in terms of their cost-sensitive counterparts.

(4.1.3∗7) Whenever the semantic domain $\mathscr{C}$ has coproducts (*e.g.* **Set**), we also have sums as a value type in a call-by-push-value language. Computational sums $X + Y$ do not play a role in this dissertation, but one may choose to include them under some mild assumptions about the monad $\mathbb{T} = \mathsf{UF} : \mathscr{C} \to \mathscr{C}$ associated with a computational effect (such as if $\mathbb{T}$ preserves filtered colimits [76]).

✎ **(4.1.3∗8)** The call-by-push-value decomposition of the programming language **PCF** for higher-order recursion can be defined as the following inductive family:

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{nat}} \qquad \frac{\Gamma \vdash v : \mathsf{nat}}{\Gamma \vdash \mathsf{suc}(v) : \mathsf{nat}}$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \qquad \Gamma \vdash e_0 : X \qquad \Gamma, z : \mathsf{nat} \vdash e_1 : X}{\Gamma \vdash \mathsf{ifz}(e, e_0, z.e_1) : X} \qquad \frac{\Gamma, x : A \vdash e : X}{\Gamma \vdash \lambda x.e : A \to X}$$

$$\frac{\Gamma \vdash e : A \to X \qquad \Gamma \vdash e_1 : A}{\Gamma \vdash e\ e_1 : X} \qquad \frac{\Gamma, x : \mathsf{U}X \vdash e : X}{\Gamma \vdash \mathsf{fix}(x.e) : X} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{ret}(a) : \mathsf{F}A}$$

$$\frac{\Gamma \vdash e : \mathsf{F}A \qquad \Gamma, a : A \vdash e_1 : X}{\Gamma \vdash \mathsf{bind}(e, a.e_1) : X}$$

(4.1.3∗9) To extend the basic call-by-push-value language of **(4.1.3∗8)** with an abstract cost effect, we may add the following computation form:

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathsf{step}^c(e) : X}$$

The intuition is that $\mathsf{step}^c(e)$ will incur c units of cost and behave like the computation e.

4.2. CALF: A COST-AWARE LOGICAL FRAMEWORK

(4.2∗1) In this section we will consolidate the discussion so far into a single dependent type theory dubbed **calf** (a **c**ost-**a**ware **l**ogical **f**ramework) for integrating functional and cost-sensitive reasoning in which cost is treated as an abstract effect.

4.2.1. Defining type theories using logical frameworks.

(4.2.1∗1) A type theory such as **calf** is a complex mathematical object to fully specify — the discussion in Section 2.1 illustrates only a small fraction of the rules typically required to give a rigorous definition of a type theory. To simplify the process of defining new type theories, we take advantage of the fact that everything difficult about defining the syntax of dependent type theory can be "factored out" once and for all into a *meta*-type theory called a *logical framework* [54, 50] that can be used to define object-type theories (such as **calf**).

☝ **(4.2.1∗2)** The process of defining type theories in logical frameworks is analogous to that of defining algebraic theories in a *doctrine*; for example, the theory of a group is an object theory of the doctrine of finite product theories.

(4.2.1∗3) Concretely, we work in a logical framework (LF) with a universe of judgments **Jdg** closed under dependent product, dependent sum, and extensional equality; the reader may find

$$\begin{aligned}
\mathbb{C} &: \mathbf{Jdg} \\
0 &: \mathbb{C} \\
+ &: \mathbb{C} \to \mathbb{C} \to \mathbb{C} \\
\sqsubseteq &: \mathbb{C} \to \mathbb{C} \to \mathbf{Jdg} \\
\mathsf{costMon} &: \mathsf{isOrderedMonoid}(\mathbb{C}, 0, +, \sqsubseteq) \\
\mathsf{step} &: \{X : \mathsf{tp}^-\}\ \mathbb{C} \to \mathsf{tm}^{\ominus}(X) \to \mathsf{tm}^{\ominus}(X) \\
\mathsf{step}_0 &: \{X, e\}\ \mathsf{step}^0(e) = e \\
\mathsf{step}_+ &: \{X, e, c_1, c_2\} \\
&\quad \mathsf{step}^{c_1}(\mathsf{step}^{c_2}(e)) = \mathsf{step}^{c_1+c_2}(e)
\end{aligned}$$

$$\begin{aligned}
\mathsf{tp}^+, \mathsf{tp}^- &: \mathbf{Jdg} \\
\mathsf{tm} &: \mathsf{tp}^+ \to \mathbf{Jdg} \\
\mathsf{U} &: \mathsf{tp}^- \to \mathsf{tp}^+ \\
\mathsf{F} &: \mathsf{tp}^+ \to \mathsf{tp}^- \\
\mathsf{tm}^{\ominus}(X) &:= \mathsf{tm}(\mathsf{U}X) \\
\mathsf{ret} &: (A : \mathsf{tp}^+, a : \mathsf{tm}(A)) \to \mathsf{tm}^{\ominus}(\mathsf{F}A) \\
\mathsf{bind} &: \{A : \mathsf{tp}^+, X : \mathsf{tp}^-\}\ \mathsf{tm}^{\ominus}(\mathsf{F}A) \to \\
&\quad (\mathsf{tm}(A) \to \mathsf{tm}^{\ominus}(X)) \to \mathsf{tm}^{\ominus}(X)
\end{aligned}$$

$$\begin{aligned}
\P &: \mathbf{Jdg} \\
\P/\mathsf{uni} &: \{u, v : \P\}\ u = v \\
\mathsf{step}/\P &: \{X, e, c\}\ \P \to (\mathsf{step}^c(e) = e)
\end{aligned}$$

$$\begin{aligned}
\mathsf{eq} &: (A : \mathsf{tp}^+) \to \mathsf{tm}(A) \to \mathsf{tm}(A) \to \mathsf{tp}^+ \\
\mathsf{self} &: \{A, a, b\}\ \mathsf{tm}(\mathsf{eq}_A(a, b)) \cong (a =_{\mathsf{tm}(A)} b)
\end{aligned}$$

$$\begin{aligned}
\mathsf{nat} &: \mathsf{tp}^+ \\
\mathsf{zero} &: \mathsf{tm}(\mathsf{nat}) \\
\mathsf{suc} &: \mathsf{tm}(\mathsf{nat}) \to \mathsf{tm}(\mathsf{nat}) \\
\mathsf{rec} &: (n : \mathsf{tm}(\mathsf{nat})) \to \\
&\quad (X : \mathsf{tm}(\mathsf{nat}) \to \mathsf{tp}^-) \to \mathsf{tm}^{\ominus}(X(\mathsf{zero})) \to \\
&\quad ((n : \mathsf{tm}(\mathsf{nat})) \to \mathsf{tm}^{\ominus}(X(n)) \to \\
&\qquad \mathsf{tm}^{\ominus}(X(\mathsf{suc}(n)))) \to \mathsf{tm}^{\ominus}(X(n))
\end{aligned}$$

Figure 4.1: Equational presentation of **calf** as a signature $\Sigma_{\mathbf{calf}}$ in the logical framework. Here the type isOrderedMonoid encodes all the structure of an preordered monoid. We write $A \cong B$ for a framework-level isomorphism between judgments.

the theoretical justifications for this logical framework in Gratzer and Sterling [43] and Harper [50] and examples of object theories defined in the LF in Niu et al. [90], Sterling and Harper [122], and Grodin et al. [46]. An object theory in the LF is specified as follows.

1. Judgments are declared as constants ending in **Jdg**.
2. Binding and scope is handled by the framework-level dependent product $(x : X) \to Y(x)$.
3. Equations between object-level terms are specified by constants ending in the framework-level equality type $x_1 =_X x_2$.

The core constructs of **calf** are displayed in Fig. 4.1.

In addition to the call-by-push-value structure presented in Fig. 4.1, **calf** is also equipped with ordinary dependent product and sum types, which means that we may carry out ordinary mathematical arguements as outlined in Section 2.4. In contrast to the more general situation described in Section 4.1, we impose the order relation on the cost monoid $(\mathbb{C}, 0, +)$ explicitly as part of the parameterization of **calf**. The reason is that in practice one is not necessarily only interested in derived prefix relation on a monoid; for instance when considering the *parallel* complexity of programs (Section 4.3) it would be sensible to consider both the order relation tracking both the parallel and sequential cost and another order relation tracking only the parallel cost.

✎ **(4.2.1∗4)** To illustrate programming in **calf**, we may port the insertion sort algorithm from **(4.1∗9)** to the call-by-push-value setting:

$$\begin{aligned}
&\mathit{insert} : \mathsf{tm}(A) \to \mathsf{tm}(\mathsf{list}(\mathsf{nat})) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{list}(\mathsf{nat})))\\
&\mathit{insert}(x, \mathsf{nil}) = \mathsf{ret}([x])\\
&\mathit{insert}(x, y :: l) = \mathsf{bind}(\mathit{lt}(x, y), \lambda b.\mathsf{if}(b, \mathsf{bind}(\mathit{insert}(x, l), \lambda r.\mathsf{ret}(y :: r)), \mathsf{ret}(x :: y :: l)))
\end{aligned}$$

$$\begin{aligned}
&\mathit{insertionSort} : \mathsf{tm}(\mathsf{list}(\mathsf{nat})) \to \mathsf{tm}^{\ominus}(\mathsf{Flist}(\mathsf{nat}))\\
&\mathit{insertionSort}(\mathsf{nil}) = \mathsf{nil}\\
&\mathit{insertionSort}(x :: l) = \mathsf{bind}(\mathit{insertionSort}(l), \lambda l'.\, \mathit{insert}(x, l'))
\end{aligned}$$

Observe that because the type of computations $\mathsf{F}A$ does not expose the implementation of cost profiling, the only way to sequence computations is by using bind, which ensures that cost is correctly aggregated.

4.2.2. Interactive cost refinement in calf.

(4.2.2∗1) Using the structure of a preordered monoid, we may conjecture that a computation $e : \mathsf{tm}^{\ominus}(\mathsf{F}A)$ is bounded by $c : \mathbb{C}$ if $e = \mathsf{step}^{c'}(\mathsf{ret}(a))$ for some $c' \sqsubseteq c$ and $a : \mathsf{tm}(A)$. While this is a perfectly sensible definition, experience suggests it is more natural to replace ordinary inequality $\sqsubseteq$ with the *restricted inequality* $\bigcirc(c' \sqsubseteq c)$. Observe that this does not trivialize cost bounds because $\mathbb{C}$ is not necessarily sealed (the axioms only ensure that the cost *effect* step^c is trivialized in the functional phase). Thus the cost bound predicate in **calf** is defined as follows.

$$\mathsf{IsBounded}\,(A; e; c) = \Sigma_{a:A}.\Sigma_{d:\mathbb{C}}(e = \mathsf{step}^d(\mathsf{ret}(a))) \times \bigcirc(d \sqsubseteq c)$$

The use of the restricted inequality in the $\mathsf{IsBounded}$ refinement reflects the intuition that "costs don't have cost". More importantly, this arrangement grants one access to the functional phase and the *purely functional* properties therein when proving cost refinements, which is essential for analyses of algorithms that depend on behavioral invariants of data structures. For instance, the cost analysis of insertion sort outlined in **(4.1.1∗7)** depends on knowing the invariant that *sorting preserves length*, a fact that follows from the correctness of sorting, which as we explained in **(4.1.2∗2)** and **(4.1.2∗3)**, is most naturally stated in the functional phase.

↗ **(4.2.2∗2)** There have been significant improvements to the original theory of cost bounds in **calf** presented in this dissertation. In particular, Grodin et al. has posited that the idea of abstracting a numerical cost bound from a program should be understood as a misguided relic of classical cost analysis that becomes nearly intractible in the setting of higher-order functions and computational effects other than cost. The central thesis of *op. cit.* is that a cost bound is just another program and that cost bound analysis should be recast in terms of *program inequalities* $e \sqsubseteq e'$ (as opposed to mere *numerical* inequalities). The resulting theory, dubbed **decalf**, supports

$$\textsc{Return}\ \frac{}{\mathsf{IsBounded}\,(A; \mathsf{ret}(a); 0)} \qquad \textsc{Step}\ \frac{\mathsf{IsBounded}\,(A; e; c)}{\mathsf{IsBounded}\,\big(A; \mathsf{step}^{d}(e); d + c\big)}$$

$$\textsc{Bind}\ \frac{\mathsf{IsBounded}\,(A; e; c) \qquad \forall a : A.\, \mathsf{IsBounded}\,(B; f(a); d(a))}{\mathsf{IsBounded}\,(B; \mathsf{bind}(e; f); \mathsf{bind}(e; \lambda a.\, c + d(a)))} \qquad \textsc{Relax}\ \frac{\mathsf{IsBounded}\,(A; e; c) \qquad c \sqsubseteq c'}{\mathsf{IsBounded}\,(A; e; c')}$$

Figure 4.2: Cost refinement lemmas in **calf** displayed in inference rule style.

a more refined and streamlined workflow for existing algorithm analyses in **calf** and extends the story developed in this dissertation to nontrivial computational effects.

(4.2.2∗3) In Section 4.4.5 we prove that "restricted cost bounds" $\bigcirc(c \sqsubseteq c')$ are equivalent to ordinary cost bounds $c \sqsubseteq c'$ for a large class of cost monoids in the intended model of **calf**. The purpose of such a theorem is to interpret the meaning of cost bounds derived in **calf**: when we have a proof of the refinement $\mathsf{IsBounded}\,(A; e; c)$, we know that $e = \mathsf{step}^{c'}(\mathsf{ret}(a))$ for some $c' : \mathbb{C}$ and $a : \mathsf{tm}(A)$ such that $\bigcirc(c' \sqsubseteq c)$ holds. For this bound to be meaningful one needs to be able to conclude from the restricted inequality $\bigcirc(c' \sqsubseteq c)$ that the expected ordinary inequality $c' \sqsubseteq c$ also holds.

(4.2.2∗4) *Cost refinement lemmas.* **calf** admits many expected principles for reasoning about the $\mathsf{isBounded}$ refinement, and we present three representative cases in Fig. 4.2.

4.2.3. Analyzing general recursive algorithms in calf.

(4.2.3∗1) As we have discussed in **(2.5∗3)**, as a type theory **calf** is naturally a theory of *total* functions. In contrast, many common algorithms exhibit nontrivial patterns of recursion that do not conform to the simplistic syntactic checks set out in **(2.5∗4)**. Thus a major obstacle we need to overcome is the representation of (total) general recursive programs in **calf** that preserves the cost structure of the original program.

(4.2.3∗2) To this end, in **(2.5∗6)** we introduced a version of the Bove–Capretta method [18] for encoding general recursive programs in which accessibility predicates are abstracted into accessibility bounds that tracks the allowable number of recursive calls. In this section we provide a general recipe for analyzing such *clocked* programs in **calf**.

(4.2.3∗3) The notion of clocked programming as described here emerged from the work of Niu and Harper where it was observed (in a general recursive setting) that the cost bound of a program can be used as an induction principle for establishing type judgments of computational type theories **(2.2∗1)**. The benefit of this approach is that in the setting of cost analysis, cost bounds are essentially always assumed to exist, which means that as a general principle, "induction on cost bounds" does not impose any additional verification overhead.

(4.2.3∗4) To explain the general workflow of algorithm analysis in **calf**, suppose one is given an algorithm $f : A \rightharpoonup B$, conceived of as a *partial function*, along with its *cost model.* Thus one should think of f as a description/definition of an algorithm external to **calf**.

1. Define a *clocked* version of the algorithm $f_{🕒} : \mathbb{N} \to A \to B$ in which the *clock input* of type $\mathbb{N}$ represents the number of available recursive calls; when the clock is nonzero, $f_{🕒}$ follows the recursion pattern exhibited by f by decrementing the clock, and when the clock is zero, $f_{🕒}$ terminates by returning a default value or raising an exception. In the body of $f_{🕒}$ step's should be placed in accordance with the given cost model.

2. Define the the associated *clocked cost recurrence* for the clocked algorithm $f/\mathsf{bound}_{🕒} : \mathbb{N} \to A \to B$.

3. Define the *recursion depth* $f_{\mathsf{depth}} : A \to \mathbb{N}$ that bounds the number of recursive calls made by f on a given input $a : A$.

4. Obtain the *complete* algorithm and cost recurrence by instantiating the clocked program and cost recurrence respectively with the recursion depth: $f(a) = f_{🕒}(f_{\mathsf{depth}}(a))(a)$ and $f/\mathsf{bound}(a) = f/\mathsf{bound}_{🕒}(f_{\mathsf{depth}}(a))(a)$.

5. Prove that the resulting algorithm f is bounded by the cost recurrence f/bound. This process is mostly mechanical: one repeatedly applies the lemmas in **(4.2.2∗4)** to break down $\mathsf{isBounded}$ goals.

6. Characterize the recurrence f/bound by (*e.g.*) computing a closed-form solution. Usually this step represents the bulk of the work in pen-and-paper algorithm analysis.

4.3. CASE STUDIES IN CALF

(4.3∗1) The examples we have studied in **calf** do not represent the state-of-the-art in the algorithms literature but include many common algorithms found in an introductory textbook: Euclid's algorithm, sequential and parallel insertion and merge sort, and amortized analysis of batched queues. For many algorithms we have verified the best known asymptotic bound, a feat that relies crucially on the ability to express functional specifications and use ordinary mathematical reasoning in **calf**. This small collection of case studies suggests an auspicious beginning to a growing library of formally verified algorithms; in fact, **calf** has been used to rigorously develop the functional correctness and cost bounds of red black trees [75], amortized (coinductive) algorithms [45], and effectful higher-order programs [46].

4.3.1. Implementation of calf.

(4.3.1∗1) As discussed in **(2.1∗7)**, one of the strengths of type theories is their amenability to computerization. Indeed the ability to implement type theories as (part of) a computer program (called a *proof assistant*) so that its logical assertions may be mechanically verified is a major motivation behind the type-theoretic roots of **calf**. However, instead of a stand-alone proof assistant, **calf** has been implemented in the Agda proof assistant as a *domain specific language* (DSL). The benefit of this approach is that one may immediately reuse the existing library of mathematical facts of Agda.

(4.3.1∗2) The implementation of **calf** in Agda consists of a series of constants and equations, a fragment of which was presented in Fig. 4.1. For instance, the basic judgmental structure of **calf** may be specified by the following **Agda** postulates:

postulate

$\mathsf{mode} : \mathbf{Set}$
$\mathsf{pos} : \mathsf{mode}$
$\mathsf{neg} : \mathsf{mode}$

postulate

$\mathsf{tp} : \mathsf{mode} \to \mathbf{Set}$
$\mathsf{tm}^{+} : \mathsf{tp}\ \mathsf{pos} \to \mathbf{Set}$

postulate

$\mathsf{F} : \mathsf{tp}\ \mathsf{pos} \to \mathsf{tp}\ \mathsf{neg}$
$\mathsf{U} : \mathsf{tp}\ \mathsf{neg} \to \mathsf{tp}\ \mathsf{pos}$

We can think of the **Agda** implementation of **calf** constitutes an algebra of the LF signature of **calf** in which the **Agda** universe **Set** plays the role of the universe of judgments **Jdg**.

(4.3.1∗3) The equational theory of **calf** is encoded by means of rewrite rules [25]. For instance we may implement the computation rule for bind as follows.

postulate

$\mathsf{bind/ret} : \{A, X\}\ \{v : \mathsf{tm}(A)\}\ \{f : (x : \mathsf{tm}(A)) \to \mathsf{tm}^{\ominus}(X)\}\ \mathsf{bind}(\mathsf{ret}(v); f) \equiv f(v)$

{-# REWRITE bind/ret #-}

Here the REWRITE pragma tells Agda to treat bind/ret as an actual Agda equality; in particular bind/ret is treated as a rewrite rule that always replaces the expression to the left of ≡ with the expression to the right.

(4.3.1∗4) Following the description of **(4.2.2∗1)**, the data associated with cost bounds is naturally captured by a record type in the Agda encoding of **calf**:

record $\mathsf{IsBounded}(A : \mathsf{tp}^{+})(e : \mathsf{tm}^{\ominus}(\mathsf{F}A))(c : \mathbb{C})$: **Set where**

$\mathsf{result} : \mathsf{tm}(A)$
$c' : \mathbb{C}$
$\mathsf{hyp/bounded} : \bigcirc(c' \sqsubseteq c)$
$\mathsf{hyp/eq} : e \equiv \mathsf{step}^{c'}(\mathsf{ret}(\mathsf{result}))$

4.3.2. Example: Euclid's algorithm in calf.

(4.3.2∗1) In this section we put everything developed so far into practice by outlining how one may verify a tight cost bound for Euclid's algorithm.

(4.3.2∗2) As indicated in the recipe from **(4.2.3∗4)**, the analysis of every algorithm begins with the definition of the cost model. In Euclid's algorithm the cost model is the number modulus operations. In **calf** is this specified by an instrumented version of the modulus operation that incurs unit cost:

$$\begin{aligned}&\mathit{mod}_{\mathsf{inst}} : \mathsf{tm}(\mathsf{nat}) \to \mathsf{tm}(\mathsf{nat}) \to \mathsf{tm}^{\ominus}(\mathsf{F}\ \mathsf{nat})\\ &\mathit{mod}_{\mathsf{inst}}(x, y) = \mathsf{step}^1(\mathsf{ret}(\mathit{mod}(x, y)))\end{aligned}$$

✎ **(4.3.2∗3)** One may also specify the cost model for *abstract* data types. As an example, in the analysis of sorting algorithms, it is customary to consider the comparison cost model in which the only operation that incurs cost is the comparison operation. In **calf** we may parameterize the analyses of sorting algorithms by the following *comparable* type:

$$\begin{aligned}&\textbf{record}\ \mathsf{Comparable} : \mathbf{Set}_1\ \textbf{where}\\ &\quad A : \mathsf{tp}^+\\ &\quad \sqsubseteq : \mathsf{tm}(A) \to \mathsf{tm}(A) \to \mathbf{Set}\\ &\quad \sqsubseteq_{\mathsf{dec}} : \mathsf{tm}(A) \to \mathsf{tm}(A) \to \mathsf{tm}^{\ominus}(\mathsf{Fbool})\\ &\quad \sqsubseteq_{\mathsf{dec}}\ /\ \sqsubseteq : \{x, y, b\}\ \to \bigcirc((x \sqsubseteq_{\mathsf{dec}} y) \equiv \mathsf{ret}(b) \to \mathsf{Reflects}\ (x \sqsubseteq y)\ b)\\ &\quad \sqsubseteq\ /\mathsf{ord} : \mathsf{isTotalOder}\ \sqsubseteq\\ &\quad \sqsubseteq_{\mathsf{dec}}\ /\mathsf{cost} : (x, y : \mathsf{tm}(A)) \to \mathsf{IsBounded}\ \mathsf{bool}\ (x \sqsubseteq_{\mathsf{dec}} y)\ 1\end{aligned}$$

In other words a comparable type is a type A equipped with a total ordering relation $\sqsubseteq$. To program with comparable types we also need the ordering to be decidable, which is encoded above as $\sqsubseteq_{\mathsf{dec}}$. Because the comparison cost model dictates that the comparison operation is unit cost, we require a field $\sqsubseteq_{\mathsf{dec}}\ /\mathsf{cost}$ to record this fact using the $\mathsf{IsBounded}$ type defined in **(4.3.1∗4)**. Lastly, the field $\sqsubseteq_{\mathsf{dec}}\ /\ \sqsubseteq$ indicates that $\sqsubseteq_{\mathsf{dec}}$ is a decision procedure for $\sqsubseteq$, *i.e.* the $x \sqsubseteq_{\mathsf{dec}} y$ computes the value $\mathsf{tt} : \mathsf{bool}$ if and only if $x \sqsubseteq y$ holds. Here we observe another use of the restriction modality $\bigcirc$: because the decision procedure $\sqsubseteq_{\mathsf{dec}}$ is a computation with nontrivial cost, we descend to the functional phase to state its correctness specification.

(4.3.2∗4) Similar to the approach taken in **(2.5∗6)**, we may define a clocked version of Euclid's algorithm in **calf**:

$$\begin{aligned}&\mathit{gcd}_{\text{🕓}} : \mathsf{tm}(\mathsf{nat}) \to \mathsf{tm}(\mathsf{nat}^2) \to \mathsf{tm}^{\ominus}(\mathsf{F}\ \mathsf{nat})\\ &\mathit{gcd}_{\text{🕓}}(\mathsf{zero}, x, y) = \mathsf{ret}(x)\\ &\mathit{gcd}_{\text{🕓}}(\mathsf{suc}(k), x, \mathsf{zero}) = \mathsf{ret}(x)\\ &\mathit{gcd}_{\text{🕓}}(\mathsf{suc}(k), x, \mathsf{suc}(y)) = \mathsf{bind}(\mathit{mod}_{\mathsf{inst}}(x, \mathsf{suc}(y)), \lambda r.\mathit{gcd}_{\text{🕓}}(k, \mathsf{suc}(y), r))\end{aligned}$$

Recall that $\mathit{mod}_{\mathsf{inst}}$ is the instrumented modulus that encodes the cost model for Euclid's algorithm. Here the first argument to $\mathit{gcd}_{\text{🕓}}$ is a clock parameter that ticks down at each recursive call of Euclid's algorithm; when the clock parameter is zero $\mathit{gcd}_{\text{🕓}}$ simply returns the first real argument. In other words $\mathit{gcd}_{\text{🕓}}(k, x, y)$ is the k-approximation of $\gcd(x, y)$ in which Euclid's algorithm is only allowed to make k recursive calls.

(4.3.2∗5) Does the clocked program $\mathit{gcd}_{\text{🕓}}$ satisfies the behavioral specification of Euclid's algorithm, *i.e.* does $\mathit{gcd}_{\text{🕓}}$ compute the gcd? We already observed that $\mathit{gcd}_{\text{🕓}}(k, x, y)$ only computes k-approximations of $\gcd(x, y)$. However, if the approximation level k (*i.e.* number of recursive calls) is sufficient, then it should be the case that $\mathit{gcd}_{\text{🕓}}(k, x, y)$ actually computes $\gcd(x, y)$. To this end, we define the *recursion depth* of Euclid's algorithm, which is a function that computes the necessary approximation level for any input to Euclid's algorithm:

$$
\begin{aligned}
&gcd_{\mathsf{depth}} : \mathsf{tm}(\mathsf{nat}^2) \to \mathsf{tm}(\mathsf{nat}) \\
&gcd_{\mathsf{depth}}(x,y) = \begin{cases} \mathsf{zero} & \text{if } y = \mathsf{zero} \\ \mathsf{suc}(gcd_{\mathsf{depth}}(y, mod(x,y))) & \text{o.w.} \end{cases}
\end{aligned}
$$

Note that gcd_{depth} is a specification of a function by cases: because we do not need to track the cost of computing the recursion depth, gcd_{depth} may be defined however convenient.[3]

(4.3.2*6) We may now instantiate the clocked algorithm $gcd_{\text{🕓}}$ by the recursion depth gcd_{depth}: define $gcd(x,y) \coloneqq gcd_{\text{🕓}}((gcd_{\mathsf{depth}}(x,y),x,y)$. We may prove that gcd computes the gcd, a behavioral specification that is naturally expressed as the following equations by means of the restriction modality:

$$\bigcirc(gcd\, x\ \mathsf{zero} = \mathsf{ret}(x)) \tag{1}$$

$$\bigcirc(gcd\, x\ (\mathsf{suc}(y)) = gcd\ (\mathsf{suc}(y))\ (mod\, x\ \mathsf{suc}(y))) \tag{2}$$

(4.3.2*7) The method of recurrence relations in traditional presentations of algorithm analysis is divided into the two expected stages in **calf**: we extract a cost recurrence f/bound from the algorithm and compute a closed-form formula ϕ for the cost recurrence. Each of these steps has an associated proof obligation: we have to show that f/bound is indeed a cost bound for the algorithm and that ϕ is an upper bound for f/bound. Recall from **(4.2.3*4)** that a cost recurrence is a function that assigns a cost to each clock and input of the algorithm. In the case of Euclid's algorithm we have the following cost recurrence associated to $gcd_{\text{🕓}}$:

$$
\begin{aligned}
&gcd/\mathsf{bound}_{\text{🕓}} : \mathsf{tm}(\mathsf{nat}) \to \mathsf{tm}(\mathsf{nat}^2) \to \mathsf{tm}(\mathsf{nat}) \\
&gcd/\mathsf{bound}_{\text{🕓}}(\mathsf{zero}, x, y) = \mathsf{zero} \\
&gcd/\mathsf{bound}_{\text{🕓}}(\mathsf{suc}(k), x, y) = \begin{cases} \mathsf{zero} & \text{if } y = \mathsf{zero} \\ \mathsf{suc}(gcd/\mathsf{bound}_{\text{🕓}}(k, y, mod(x,y))) & \text{o.w.} \end{cases}
\end{aligned}
$$

Similar to the case for the recursion depth **(4.3.2*5)**, we do not track the cost computing the cost recurrence, which is indicated by the fact that the codomain of $gcd/\mathsf{bound}_{\text{🕓}}$ is the value type nat as opposed to the computation type $\mathsf{F}\ \mathsf{nat}$.

(4.3.2*8) We may prove that $gcd/\mathsf{bound}_{\text{🕓}}$ is a cost bound for $gcd_{\text{🕓}}$, which is expressed by the following theorem:

$$gcd_{\text{🕓}}/\mathsf{bound} : (k, x, y : \mathsf{tm}(\mathsf{nat})) \to \mathsf{IsBounded}\ \mathsf{nat}\ (gcd_{\text{🕓}}\ k\ x\ y)\ (gcd/\mathsf{bound}_{\text{🕓}}\ k\ x\ y)$$

The proof of $gcd_{\text{🕓}}/\mathsf{bound}$ is entirely mechanical: the user simply breaks down the overall $\mathsf{IsBounded}$ proof goal and fulfills the generated sub-goals using the syntax-directed cost refinement lemmas (depicted in Fig. 4.2). In fact this step is taken to be so obvious that often no proof is given in textbook presentations of algorithm analysis.

(4.3.2*9) The last and usually most difficult step in algorithm analysis is to compute a closed-form solution or otherwise informative bound to the cost recurrence. This step is also the place where **calf** shines as a mathematical domain for reasoning about cost bounds. For instance, we may prove a very precise bound on the cost recurrence $gcd/\mathsf{bound}\ x\ y \coloneqq gcd/\mathsf{bound}_{\text{🕓}}\ (gcd_{\mathsf{depth}}\ x\ y)\ x\ y$

[3]In Agda we define gcd_{depth} using well-founded induction on the last argument.

that is known to be asymptotically tight.[4] Let $\mathsf{Fib} : \mathbb{N} \to \mathbb{N}$ be the fibonacci sequence, and let $\mathsf{Fib}^{-1} : \mathbb{N} \to \mathbb{N}$ be the function characterized by the equation $\mathsf{Fib}^{-1}(x) = \max \{i \mid \mathsf{Fib}(i) \sqsubseteq x\}$. We have the following **calf** theorem:

$$\mathit{gcd}/\mathsf{bound}/\mathsf{bound}/\mathsf{isBound} : (x, y : \mathsf{tm}(\mathsf{nat})) \to (x > y) \to \mathit{gcd}/\mathsf{bound}\ x\ y \sqsubseteq 1 + \mathsf{Fib}^{-1}(x)$$

In conjunction with **(4.3.2∗8)**, we obtain the following cost bound for *gcd*:

$$\mathit{gcd}/\mathsf{isBounded} : (x, y : \mathsf{tm}(\mathsf{nat})) \to (x > y) \to \mathsf{IsBounded}\ \mathsf{nat}\ (\mathit{gcd}\ x\ y)\ (1 + \mathsf{Fib}^{-1}(x))$$

Thus the number of modulus operations used in Euclid's algorithm never exceeds the quantity $1 + \mathsf{Fib}^{-1}(x)$, where x is the larger of the two inputs. This means that asymptotically the computational cost of *gcd* is logarithmic in the input (*i.e.* proportional to the *size* of the input).

∗ ∗ ∗

(4.3.2∗10) As mentioned in **(4.2.2∗1)**, it is often necessary to access the functional phase of **calf** when proving bounds on cost recurrences. For instance, when computing the closed-form solution to the cost recurrence of *insertionSort* from **(4.2.1∗4)**, we would like to have access to the theorem $\mathit{insertionSort}/\mathsf{correct} : \mathsf{IsSort}\ \mathit{insertionSort}$ stating that *insertionSort* is a sorting algorithm, where IsSort is the following family.

$$\begin{aligned}&\mathsf{IsSort} : (\mathsf{tm}(\mathsf{list}(A)) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{list}(A)))) \to \mathbf{Set}\\ &\mathsf{IsSort}\ f\ = (l : \mathsf{tm}(\mathsf{list}(A))) \to \bigcirc(\Sigma_{l' : \mathsf{tm}(\mathsf{list}(A))}.f\ l \equiv l' \times \mathsf{SortedOf}\ l\ l')\end{aligned}$$

In the above $\mathsf{SortedOf}\ l\ l'$ is the *restriction-modal* proposition characterizing the correctness of sorting as discussed in **(4.1.2∗2)**. In particular, one relies on the fact that every map f satisfying $\mathsf{IsSort}(f)$ is a length-preserving function (either by definition or via another restriction-modal proposition), which is used in the inductive case of proof for the closed form characterization of the cost recurrence of *insertionSort*. Observe that the predicate IsSort is also restriction-modal because the cost incurred by the candidate sorting program is irrelevant from the perspective of pure functional correctness. Consequently we may relate $\mathit{insertionSort}/\mathsf{bound}$ (the cost recurrence associated to *insertionSort*) to its closed-form solution in the functional phase, as indicated in the following theorem.

$$\mathit{insertionSort}/\mathsf{bound}/\mathsf{isBound} : (l : \mathsf{tm}(\mathsf{list}(A))) \to \bigcirc(\mathit{insertionSort}/\mathsf{bound}\ l \sqsubseteq |l|^2)$$

4.3.3. Parallelism in calf.

(4.3.3∗1) Parallelism arises naturally in the setting of **calf** via an equational presentation of the profiling semantics of Blelloch and Greiner [17]. Here we present a version adapted from Harper [51] in which it is observed that the source of parallelism can be isolated to the treatment of *pairs* of computations: a parallel computation of $A \times B$ is furnished by a new computation form & called *joins* that conjoins two independent computations of A and B:

$$\& : \{A, B : \mathsf{tp}^{+}\}\ \mathsf{tm}^{\ominus}(\mathsf{F}A) \to \mathsf{tm}^{\ominus}(\mathsf{F}B) \to \mathsf{tm}^{\ominus}(\mathsf{F}(A \times B))$$

[4]We have not verified in Agda/**calf** that it is indeed the tightest bound possible. But **calf** allows the user to prove these results if desired.

One may think of a term $e \,\&\, f$ as a computation in which e and f are evaluated simultaneously.

(4.3.3∗2) Blelloch and Greiner [17] characterizes the complexity of a program in terms of pairs (w, s) in which w represents the *work* or sequential cost and s represents *span* or parallel cost of a computation. We call pairs (w, s) the *cost graph* of a computation. The *cost graph semantics* of parallel programs associates to a program e a cost graph. For instance, suppose that we have computations e_1 and e_2 with cost graphs (w_1, s_1) and (w_2, s_2) respectively. Then we have that the sequential composition of e_1 and e_2 is assigned the cost graph $(w_1 + w_2, s_1 + s_2)$ and the parallel composition of e_1 and e_2 is assigned the cost graph $(w_1 + w_2, \max(s_1, s_2))$. Thus the work of a computation is the total resource usage and the span is the resource usage along the *critical path*, defined to be the thread within a computation with the highest resource usage. In the particular case of the resource of time, we may think of work as the total number of CPU cycles and span as the actual time elapsed during computation.

(4.3.3∗3) Accordingly, in **calf** the cost graph is encoded by the following structure called the *parallel cost monoid*:

$$\mathbb{C} \coloneqq (\mathbb{N}^2, \oplus, (0,0), \sqsubseteq_{\mathbb{N}^2})$$

In the above $\oplus$ and $\sqsubseteq_{\mathbb{N}^2}$ are component-wise extensions of addition and $\sqsubseteq$. Parallel cost composition is then implemented by the operation $(w_1, s_1) \otimes (w_2, s_2) \coloneqq (w_1 + w_2, \max(s_1, s_2))$ that takes the sum of the works and max of the spans. This provides the required structure to assemble the cost of a completed parallel pair:

$$\&_{\mathsf{join}} : \{A, B, c_1, c_2, a, b\} \;\; (\mathsf{step}^{c_1}(\mathsf{ret}(a))) \,\&\, (\mathsf{step}^{c_2}(\mathsf{ret}(b))) = \mathsf{step}^{c_1 \otimes c_2}(\mathsf{ret}((a, b)))$$

✎ **(4.3.3∗4)** Using the parallel cost monoid, we may derive the cost bound for *parallel* merge sort in **calf**. Assume as in **(4.3.2∗3)** that we work relative to the comparison cost model for sorting:

record $\mathsf{Comparable} : \mathbf{Set}_1$ **where**

- $A : \mathsf{tp}^+$
- $\sqsubseteq : \mathsf{tm}(A) \to \mathsf{tm}(A) \to \mathbf{Set}$
- $\sqsubseteq_{\mathsf{dec}} : \mathsf{tm}(A) \to \mathsf{tm}(A) \to \mathsf{tm}^{\ominus}(\mathsf{Fbool})$
- $\sqsubseteq_{\mathsf{dec}} / \sqsubseteq : \{x, y, b\} \;\to \bigcirc((x \sqsubseteq_{\mathsf{dec}} y) \equiv \mathsf{ret}(b) \to \mathsf{Reflects}\ (x \sqsubseteq y)\ b)$
- $\sqsubseteq /\mathsf{ord} : \mathsf{isTotalOder}\ \sqsubseteq$
- $\sqsubseteq_{\mathsf{dec}} /\mathsf{cost} : (x, y : \mathsf{tm}(A)) \to \mathsf{IsBounded}\ \mathsf{bool}\ (x \sqsubseteq_{\mathsf{dec}} y)\ (1, 1)$

The only difference from the model in **(4.3.2∗3)** is that the cost of the comparison operator is given by a cost graph.

(4.3.3∗5) Similar to insertion sort, we may define a version of merge sort $\mathit{msortPar} : \mathsf{list}(A) \to \mathsf{F}(\mathsf{list}(A))$ for a comparable type A in which both the sorting and merging step are computed in parallel by means of the join operator. The definition of *msortPar* may be found in the **calf** library.[5]

☐ **(4.3.3∗6)** For all $l : \mathsf{list}(A)$, we have that the following proposition holds in **calf**:

$$\mathsf{IsBounded}\Big(\mathsf{list}(A); \mathit{msortPar}(l); \Big(\lceil \log_2(|l| + 1) \rceil^2 \cdot |l|, \lceil \log_2(|l| + 1) \rceil^3\Big)\Big).$$

[5]`https://github.com/jonsterling/agda-calf/blob/v1.0.0/src/Examples/Sorting/Parallel/MergeSortPar.agda`.

4.4. MODEL OF CALF

(4.4∗1) In this section we explicate the semantics of **calf** in a topos model of the FC-phase distinction in the sense of **(4.1∗2)**.

(4.4∗2) Fix a topos $\mathcal{E}$ and universe $\mathcal{U}$ in $\mathcal{E}$. A $\mathcal{U}$-small *model* of a LF signature Σ is defined to be a Σ-algebra in which the type of judgments **Jdg** is defined to be $\mathcal{U}$. Viewing an LF signature as a theory in the doctrine of locally Cartesian closed categories (lcccs), such an algebra extends uniquely to a lccc functor from the *free* lccc on Σ to $\mathscr{E}$.

(4.4∗3) In this section we fix a topos model $(\mathscr{E}, \phi)$ of the FC-phase distinction and an internal preordered monoid $(\mathbb{C}, 0, +, \sqsubseteq)$ in $\mathscr{E}$ such that $\mathbb{C}$ is *restriction-modal, i.e.* $\bigcirc\mathbb{C} \cong \mathbb{C}$.

4.4.1. Judgmental structure.

(4.4.1∗1) Recall from Section 3.2.4 the notion of an algebra over a monad. A *cost algebra* is an algebra for the writer monad $\mathbb{C} \times -$. As mentioned in **(4.1.3∗5)**, the coherence conditions on a cost algebra ensures that the structure map commutes with the monoid structure.

(4.4.1∗2) Fixing universe levels $\alpha < \beta$, we may define a $\mathcal{U}_\beta$-small model of **calf** based on the discussion of the cost algebra semantics of the cost effect. The judgments of **calf** are defined as follows.

$$
\begin{aligned}
&\mathsf{tp}^+ = \mathcal{U}_\alpha \\
&\mathsf{tp}^- = \mathsf{Alg}_{\mathcal{U}_\alpha}(\bullet\mathbb{C} \times -) \\
&\mathbb{C} = \mathbb{C} \\
&\P = \phi \\
&\mathsf{tm}(A) = A
\end{aligned}
$$

In other words, we interpret a value type A to be a type in $\mathscr{E}$ and a computation type X as a $\mathcal{U}_\alpha$-small cost algebra. The proposition ¶ representing the functional phase is interpreted by the distinguished proposition ϕ in $\mathscr{E}$. Observe that we *seal* the cost monoid $\mathbb{C}$ in the interpretation of computation types but not the interpretation of the judgment $\mathbb{C}$; this is because although the cost structure of *programs* is trivialized in the functional phase, $\mathbb{C}$ itself should be a restriction modal type because we employ *functional* inequalities $\bigcirc(c \sqsubseteq d)$ in the $\mathsf{IsBounded}$ refinement **(4.2.2∗1)**.

(4.4.1∗3) The basic call-by-push-value structure is interpreted by the free-forgetful adjunction between cost algebras and sets:

$$
\begin{aligned}
&\mathsf{F}A = \mathsf{free}_{\bullet\mathbb{C}\times-}(A) \\
&\mathsf{U}X = |X|
\end{aligned}
$$

4.4.2. Type structure.

(4.4.2∗1) Value types are interpreted as the corresponding types in $\mathscr{E}$. For computation types, we need to verify that they exhibit cost algebra structures.

(4.4.2∗2) For example, given a family of cost algebras $(X(a), \alpha_{X(a)})$ over any set A, the dependent product $\Pi_{a:A}.X(a)$ of the family is a cost algebra, with the algebra map defined pointwise.

■ **(4.4.2∗3)** To show that $\Pi_{a:A}.X(a)$ is a cost algebra, we need to define an algebra map $\alpha : \bullet\mathbb{C} \times \Pi_{a:A}.X(a) \to \Pi_{a:A}.X(a)$. We define $\alpha(c, f, a) = \alpha_X(c, a)$. The coherence conditions of **(3.2.4∗2)** follow because each set $X(a)$ is assumed to be a cost algebra.

(4.4.2∗4) Show that given a family of cost algebras $(X(a), \alpha_{X(a)})$ over any set A, the dependent sum $\Sigma_{a:A}.X(a)$ is also a cost algebra.

4.4.3. Sequential composition on the free algebra.

(4.4.3∗1) Sequential composition in a call-by-push-value language can be defined uniformly for any monad $\mathbb{T}$. In this section we fix such a monad.

(4.4.3∗2) For the unit of sequential composition, we need to define a map $\mathsf{ret} : A \to |\mathsf{F}A|$. By definition of free algebras **(3.2.4∗6)**, we have that $|\mathsf{F}A| = |\mathsf{free}_{\bullet\mathbb{C}}| = \mathsf{T}(A)$, so we may take this map to be the component of the monad unit $\eta_{\mathbb{T}}$ at A.

(4.4.3∗3) For sequential composition, we need to define a map of the following type for every set A and $\mathbb{T}$-algebra X:

$$\mathsf{bind} : \mathsf{T}(A) \to (A \to |X|) \to |X|$$

By **(3.2.4∗7)**, we have that $f : A \to |X|$ corresponds to the cost algebra morphism $\overline{f} : \mathsf{free}_{\bullet\mathbb{C}\times -}(A) \to X$ determined by the function $\alpha_X T f : \mathsf{T}A \to |X|$. Therefore we just implement bind by function application.

□ **(4.4.3∗4)** For every $f : A \to X$ and $a : A$, we have that $\mathsf{bind}(\mathsf{ret}(a), f) = f(a)$.

■ **(4.4.3∗5)** Since ret is interpreted by the monad unit, this is a direct consequence of the naturality of the unit $\eta_{\mathbb{T}}$.

□ **(4.4.3∗6)** For every $e : |\mathsf{F}A|$, $f : A \to |\mathsf{F}B|$, and $g : B \to |X|$, we have that $\mathsf{bind}(\mathsf{bind}(e, f), g) = \mathsf{bind}(e, \lambda a.\mathsf{bind}(f(a), g))$.

■ **(4.4.3∗7)** Unfolding definitions, we have that $\mathsf{bind}(\mathsf{bind}(-, f), g)$ is interpreted as $\overline{g}\overline{f}$, where we write $\overline{f}$ for the backward direction of the bijection $\mathrm{Hom}_{\mathsf{Alg}(\mathbb{T})}(\mathsf{free}_{\mathbb{T}}(A), X) \cong \mathrm{Hom}_{\mathbf{Set}}(A, |X|)$. On the other hand, $\mathsf{bind}(\mathsf{bind}(-, f), g)$ is interpreted as $\overline{\overline{g}f}$, and so the result follows by considering the naturality of the hom-set bijection at the algebra morphism $\overline{g} : \mathsf{T}B \to X$:

$$\begin{tikzcd}
\mathrm{Hom}_{\mathbf{Set}}(A, \mathsf{T}B) \arrow[r] \arrow[d] & \mathrm{Hom}_{\mathsf{Alg}(\mathbb{T})}(\mathsf{free}_{\mathbb{T}}(A), \mathsf{free}_{\mathbb{T}}(B)) \arrow[d] \\
\mathrm{Hom}_{\mathbf{Set}}(A, |X|) \arrow[r] & \mathrm{Hom}_{\mathsf{Alg}(\mathbb{T})}(\mathsf{free}_{\mathbb{T}}(A), X)
\end{tikzcd}$$

4.4.4. Cost effect.

(4.4.4∗1) The cost effect $\mathsf{step}_X : \mathbb{C} \times X \to X$ is defined by first sealing the input cost and applying the algebra map of X:

$$\mathsf{step}(c, x) = \alpha_X(\eta_{\bullet -}(c), x)$$

We can easily verify that step becomes the identity map whenever ϕ holds:

$$\begin{aligned}\mathsf{step}(c,x) &= \alpha_X(\eta_{\bullet -}(c), x)\\ &= \alpha_X(*, x)\\ &= \alpha_X(\eta_{\bullet -}(0), x)\\ &= \alpha_X(\eta_{\bullet \mathbb{C}\times -}, x)\\ &= x\end{aligned}$$

Where the last equation follows from the first coherence law of cost algebras **(3.2.4∗2)**.

4.4.5. Restricted cost bounds.

☐ **(4.4.5∗1)** Any restriction-modal monoid $\mathbb{C}$ has the property that the functional prefix order relation $\bigcirc(x \sqsubseteq y)$ is equivalent to the prefix order $x \sqsubseteq y$.

■ **(4.4.5∗2)** The direction $x \sqsubseteq y \implies \bigcirc(x \sqsubseteq y)$ is evident. For the reverse direction, because $\mathbb{C}$ is restriction-modal and $\bigcirc$ preserves dependent sums and equality, we have the following equivalence:

$$\bigcirc(\Sigma_{k:\mathbb{C}}.x + k = y) \cong (\Sigma_{k:\mathbb{C}}.\eta_\bullet(x+k) =_{\bigcirc\mathbb{C}} \eta_\bullet(y))$$

Since $\eta : \mathbb{C} \to \bigcirc\mathbb{C}$ is an isomorphism, we have $x + k = y$ for some $k : \mathbb{C}$, which by definition means $x \sqsubseteq y$.

(4.4.5∗3) Proposition 4.4.5∗1 shows that the restricted inequalities $\bigcirc(x \sqsubseteq y)$ used in the definition of the IsBounded predicate of **(4.2.2∗1)** do not produce any additional cost bounds that one would not ordinarily expect (and despite the name, restricted inequalities clearly do not restrict the class of cost bounds defined using ordinary inequality). The result also extends beyond just the canonically-defined prefix order on a monoid, as long as the order relation is definable using dependent sum and equality types.

4.4.6. Main result.

☐ **(4.4.6∗1)** In every topos model of the FC-phase distinction, there is a model of **calf**.

■ **(4.4.6∗2)** By Sections 4.4.1 to 4.4.4.

⩔ **(4.4.6∗3)** The category of presheaves on $\mathbb{I}$ and the intermediate proposition u furnishes a nontrivial model of **calf**.

(4.4.6∗4) The model of **(4.4.6∗3)** corresponds to the canonical way of modeling cost structure in presheaves, which we have discussed in Section 3.1.1. In $\widehat{\mathbb{I}}$ a **calf** type is interpreted as a family $\begin{array}{c}X(1)\\ \downarrow\\ X(0)\end{array}$ in which $X(1)$ can be thought of a cost-sensitive set that *restricts* to its functional component $X(0)$. For example, we may compute the meaning of the cost effect for the cost monoid $\mathbb{N}_{\widehat{\mathbb{I}}}$:

$$\mathsf{F}A = \bullet\mathbb{N}_{\widehat{\mathbb{I}}} \times A = \begin{array}{c}\mathbb{N}\\ \downarrow\\ 1\end{array} \times \begin{array}{c}A(1)\\ \downarrow\\ A(0)\end{array} = \begin{array}{c}\mathbb{N}\times A(1)\\ \downarrow\\ A(0)\end{array}$$

A cost sensitive function f then corresponds to a square configured as follows.

$$\begin{tikzcd}
\mathbb{N} \times A(1) \arrow[r, "f_1"] \arrow[d] & \mathbb{N} \times B(1) \arrow[d] \\
A(0) \arrow[r, "f_0"'] & B(0)
\end{tikzcd} \tag{4.4.6*4*1}$$

Thus a cost-sensitive function is composed of a cost-sensitive component f_1 that lies over its purely functional component. From a global perspective, one can understand the dependence of a cost-sensitive function on its functional component as the codomain fibration $\begin{array}{c}\mathbf{Set}^{\to} \\ \downarrow \\ \mathbf{Set}\end{array}$.

(4.4.6*5) Eq. (4.4.6*4*1) is an instance of the *noninterference* property discussed in **(3.1.1*3)**. In this case, we have that the cost-sensitive component $f_1 : \mathbb{N} \times A(1) \to \mathbb{N} \times B(1)$ can never use the input cost component to compute (in a nontrivial way) the output $B(1)$. Conversely, if A and B are restriction-modal types (so that $A(1) \cong A(0)$), every $f : \mathbb{N} \times A(1) \to \mathbb{N} \times B(1)$ that is *functionally constant* (in the sense that $(\pi_1 f)(-, a) : \mathbb{N} \to B(1)$ is constant) restrict to a purely functional component $A(0) \to B(0)$.

(4.4.6*6) Noninterference is an important property because we want to be able to carry out purely functional reasoning in the absence of cost, which is not possible if the functional semantics of programs depend on cost structure.

* * *

(4.4.6*7) The formulation of a phase distinction in the sense used in this dissertation first appeared in the work of Sterling and Harper on the type-theoretic semantics of program modules and logical relations. Since then phase distinctions have been used in cost-sensitive programming and verification [90, 87, 46], metatheory of type theories [118, 116], information flow security [122], and interactive proof assistants [44].

Part II

Semantics

CHAPTER 5

Relating calf and operational cost semantics

(5∗1) We have developed the *equational* theory of cost structure and cost analysis in Chapter 4. In this chapter we develop a general method for relating this equational theory to *operational cost semantics.* As outlined in Section 0.5, a series of connections of this kind can be composed together to explain and ground the high-level mathematical semantics of programs in terms of their low-level implementations.

5.1. INTERNAL DENOTATIONAL SEMANTICS

(5.1∗1) To illustrate the key ideas of this section we work with a simple programming language, namely a version of **STLC** equipped with a base type 2 of observations. The **STLC** terms are defined by the following inductive family:

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B} \qquad \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e_1 : A}{\Gamma \vdash e\ e_1 : B} \qquad \frac{}{\Gamma \vdash \mathsf{yes} : 2}$$

$$\frac{}{\Gamma \vdash \mathsf{no} : 2}$$

We have a type tp of **STLC** types and a type family tm of **STLC** terms indexed in an **STLC** type. In the case of a call-by-push-value language such as **(4.1.3∗8)**, this generalizes to a pair of types $\mathsf{tp}^{\pm}$ to distinguish between value and computation types.

(5.1∗2) Note that to avoid excessive layers of abstractions from polluting the semantic picture, we will work directly in the internal type theory of a topos model $(\mathscr{E}, \phi)$ of the FC-distinction; this is a reasonable approach because by **(4.4.6∗1)** the results we prove can always be factored through the model of **calf** in $\mathscr{E}$.

(5.1∗3) Fix a sealed cost monoid $\mathbb{C}$. The *internal operational cost semantics* of $\mathcal{L}$ is given by a family of relations $\Downarrow_A \subseteq \mathsf{tm}(A) \times \mathbb{C} \times \mathsf{tm}(A)$. The proposition $e \Downarrow^c v$ should be read as "e evaluates to a terminal value v using c units of resource".

(5.1∗4) Commonly the operational cost semantics of a language is defined in terms of more primitive (low-level) relations $\mapsto_A \subseteq \mathsf{tm}(A) \times \mathbb{C} \times \mathsf{tm}(A)$ and $\mathsf{val}_A \subseteq \mathsf{tm}(A)$ specifying the *one-step transition relation* on programs and the *terminal* programs, respectively. In this arrangement, a program e can be thought of as representing the state of an abstract machine, and the relation

$e \xrightarrow{c} e'$ can be understood as a transition of states that additionally consumes c units of resource. A program $e \in \mathsf{val}$ is a terminal state, *i.e.* does not transition to any further states.

📖 **(5.1∗5)** Define the *reflexive-transitive closure* of $\mapsto_A$ as the smallest family $\xrightarrow{-}^* \subseteq \mathsf{tm}(A) \times \mathbb{C} \times \mathsf{tm}(A)$ closed under the following rules:

$$\frac{}{e \xrightarrow{0}^* e} \qquad\qquad \frac{e \xrightarrow{c_1} e_1 \qquad e_1 \xrightarrow{c_2}^* e_2}{e \xrightarrow{c_1+c_2}^* e_2}$$

(5.1∗6) The "small-step" semantics of **(5.1∗4)** determines a "big-step" semantics as in **(5.1∗3)** by taking the reflexive-transitive closure of $\mapsto$. More precisely, we may define $e \Downarrow^c v = (e \xrightarrow{c}^* v) \wedge (v\ \mathsf{val})$. We may also define a cost-insensitive or functional big step semantics by forgetting the cost of evaluation: $e \Downarrow v = \exists c.e \Downarrow^c v$.

(5.1∗7) The source of cost in a one-step transition $e \xrightarrow{c} e'$ is usually taken to be a single unit of resource, *i.e.* $c = 1$. For a call-by-value language, this results in the *β-reduction cost model* that is comparable to cost models commonly considered in (parallel) computational complexity [17]. We (implicitly) build this cost model into the programming language in Section 5.2, but more generally one may *parameterize* a language with an arbitrary cost model by means of a cost effect as in Section 4.1.3.

💡 **(5.1∗8)** One may also define a big-step semantics from a small-step semantics by means of a recursive function that iterates the one-step transition relation until a terminal value is reached. We exploit this fact in Chapter 8 to define the operational cost semantics of **PCF** in a setting in which the reflexive-transition closure is not available.

⚠ **(5.1∗9)** Since the notion of operational cost semantics as discussed in **(5.1∗3)** and **(5.1∗6)** only makes sense for *raw terms*, we are explicitly *not* equipping **STLC** with an equational theory. Nonetheless, one may define a standard denotational semantics of **STLC** in any topos by means of the Cartesian closed structure and the type of booleans.[1] We write $[\![-]\!]$ for both the map $\mathsf{tp} \to \mathcal{U}$ interpreting **STLC** types as types of the ambient type theory and the family of maps $(\mathsf{tm}(A) \to [\![A]\!])_{A:\mathsf{tp}}$ sending **STLC** terms to elements of the specified semantic domain; the specifics of (a generalization of) this standard interpretation is elaborated in Section 5.2.1.

📖 **(5.1∗10)** We say that the **STLC** satisfies *computational adequacy* in the sense of Plotkin [96] when for all closed programs $e, v : \mathsf{tm}(\mathsf{bool})$, we have that $[\![e]\!] = [\![v]\!]$ if and only if $e \Downarrow v$.

📖 **(5.1∗11)** In a cost-sensitive setting, the denotational semantics $[\![e]\!]$ of a term at base type consists not only of its functional meaning but also the induced cost. We say that a language satisfies *cost-sensitive computational adequacy* in the sense of Plotkin [96] when for all closed programs e of a base type, we have that $[\![e]\!] = (c, [\![v]\!])$ if and only if $e \Downarrow^c v$. In other words, we have that the denotational and operational cost semantics agree on both the functional meaning and cost structure of programs.

(5.1∗12) Although **(5.1∗11)** only gives the adequacy criterion for a total language, the techniques explained in this chapter are generalized in Chapter 8 to a more realistic programming

[1]Though as pointed out to me by Jon, one should not say the interpretation is *canonical* since without an equational theory one is not forced to *e.g.* send the function types of the **STLC** to exponentials in the semantic category.

language admiting partial functions. To illustrate the main ideas we work with the *simply-typed lambda calculus* (**STLC**) admitting only total functions.

(5.1∗13) As a meta-remark, in this chapter we will be somewhat more explicit about the mundane (in the context of this dissertation) aspects of programming languages such as substitutions and the lifting of certain constructions to substitutions/terms-in-contexts. In Chapter 8 we will work in a more relaxed manner to better focus on the semantically interesting problems and trust the reader to fill in the routine definitions.

5.2. COST-SENSITIVE COMPUTATIONAL ADEQUACY FOR THE STLC

(5.2∗1) In the following sections we fix the cost monoid $\mathbb{C} = \bullet\mathbb{N}$.

5.2.1. Denotational cost semantics.

(5.2.1∗1) To incorporate cost structure into the ordinary model of **STLC (5.1∗9)**, we further refine this interpretation by means of the call-by-push-value decomposition of call-by-value semantics [74, 66].

(5.2.1∗2) The types of **STLC** are interpreted as follows.

$$
\begin{aligned}
&[\![2]\!] = 2\\
&[\![A \to B]\!] = [\![A]\!] \to \mathbb{C} \times [\![B]\!]
\end{aligned}
$$

This interpretation extends in the evident way to an interpretation of contexts $[\![\Gamma]\!]$.

(5.2.1∗3) We write $a \leftarrow e; f(a)$ for the monadic bind operation and $\mathsf{step}^c : \mathbb{C} \times A \to \mathbb{C} \times A$ for the map $(c', a) \mapsto (c + c' +_\bullet c, a)$. We also define $\mathsf{inc} = \mathsf{step}^{1^\bullet}$, where $1^\bullet$ is the sealed cost $\eta_\bullet(1)$.

(5.2.1∗4) We define the interpretation of terms as follows.

$$
\begin{aligned}
&[\![-]\!] : \{\Gamma, A\}\ [\![\Gamma]\!] \to \mathbb{C} \times [\![A]\!]\\
&[\![x]\!](\gamma) = (0, \pi(\gamma))\\
&[\![\mathsf{yes}]\!](-) = (0, \mathsf{inl} \cdot *)\\
&[\![\mathsf{no}]\!](-) = (0, \mathsf{inr} \cdot *)\\
&[\![\lambda.e]\!](\gamma) = (0, \lambda a.[\![e]\!](\gamma, a))\\
&[\![e\ e_1]\!](\gamma) = f \leftarrow [\![e]\!](\gamma); a \leftarrow [\![e_1]\!](\gamma); \mathsf{step}^{\eta_\bullet(1)}(f\ a)
\end{aligned}
$$

In the above, we write $\pi : [\![\Gamma]\!] \to [\![A]\!]$ for the projection map induced by a variable in context $x : A \in \Gamma$.

(5.2.1∗5) Define the (open) *values* to be the following family:

$$
\frac{x \in \Gamma}{\Gamma \vdash x\ \mathsf{val}(2)}
\qquad
\frac{}{\Gamma \vdash \mathsf{yes}\ \mathsf{val}(2)}
\qquad
\frac{}{\Gamma \vdash \mathsf{no}\ \mathsf{val}(2)}
\qquad
\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x.e)\ \mathsf{val}(A \to B)}
$$

Observe that closed open values correspond to ordinary values.

□ **(5.2.1∗6)** The interpretation of general terms restricts to an interpretation of values $[\![-]\!] : \{\Gamma, A\}\ \mathsf{val}(A) \to [\![\Gamma]\!] \to [\![A]\!]$.

(5.2.1∗7) A *substitution* $s : \Delta \to \Gamma$ is a family of *open values* $\Delta \vdash v : A$ for every $x : A \in \Gamma$. Given a substitution $s : \Delta \to \Gamma$ and a term $\Gamma \vdash e : A$, we write $e[s]$ for the result of substituting e along s.[2] We write $\mathsf{Sub}(\Delta, \Gamma)$ for the collection of substitutions from Γ to Δ.

(5.2.1∗8) The interpretation of substitutions is given in a pointwise manner:

$$
\begin{aligned}
&[\![-]\!] : \{\Gamma, \Delta\}\ \mathsf{Sub}(\Delta, \Gamma) \to [\![\Delta]\!] \to [\![\Gamma]\!] \\
&[\![\cdot]\!](\delta) = * \\
&[\![s, v/x]\!](\delta) = ([\![s]\!], [\![v]\!])
\end{aligned}
$$

□ **(5.2.1∗9)** The denotational semantics is compositional, in the sense that $[\![-]\!]$ commutes with substitution: $[\![e[s]]\!] = [\![e]\!][\![s]\!]$.

■ **(5.2.1∗10)** By induction on the derivation of terms.

5.2.2. Operational cost semantics.

(5.2.2∗1) The one-step transition relation for call-by-value **STLC** is defined as the following family:

$$
\frac{e \mapsto e'}{e\ e_1 \mapsto e'\ e_1}
\qquad
\frac{e\ \mathsf{val} \qquad e_1 \mapsto e_1'}{e\ e_1 \mapsto e\ e_1'}
\qquad
\frac{e_1\ \mathsf{val}}{(\lambda x.e)\ e_1 \mapsto e[e_1/x]}
$$

Along with the restriction of **(5.2.1∗5)** to closed terms, we obtain a small-step operational cost semantics of **STLC** in which every transition step has unit cost (in the sealed cost monoid $\mathbb{C} = \bullet\mathbb{N}$).

□ **(5.2.2∗2)** The big-step cost semantics induced by Definition 5.2.2∗1 has an alternative characterization as the following inductive family:

$$
\frac{}{\mathsf{yes} \Downarrow^{0} \mathsf{yes}}
\qquad
\frac{}{\mathsf{no} \Downarrow^{0} \mathsf{no}}
\qquad
\frac{e \Downarrow^{c} \lambda x.e' \qquad e_1 \Downarrow^{c_1} v_1 \qquad e'[v_1/x] \Downarrow^{c_2} v}{e\ e_1 \Downarrow^{c + c_1 + 1^\bullet + c_2} v}
$$

5.2.3. Soundness.

□ **(5.2.3∗1)** We have that the denotational cost semantics is sound for the operational cost semantics in the following sense: if $e \Downarrow^{c} v$, then $[\![e]\!] = (c, [\![v]\!])$.

■ **(5.2.3∗2)** By induction on the derivation of the premise by the characterization in **(5.2.2∗2)**. We illustrate the case for function application. By the induction hypothesis, we have the following equations:

1. $[\![e]\!] = (c, \lambda x.[\![e']\!])$.
2. $[\![e_1]\!] = (c_1, [\![v_1]\!])$.
3. $[\![e[v_1/x]]\!] = (c_2, [\![v]\!])$.

[2]Observe that the direction of substitutions is reversed because contexts occur in a *contravariant* position relative to terms.

On the other hand, we may compute the denotational semantics, recalling from **(5.2.1∗3)** that $\mathsf{inc} = \mathsf{step}^{\eta\bullet(1)}$.

$$\begin{aligned}
[\![e\ e_1]\!] &= f \leftarrow [\![e]\!](\gamma); a \leftarrow [\![e_1]\!](\gamma); \mathsf{inc}(f\ a) \\
&= f \leftarrow (c, \lambda x.[\![e']\!]); a \leftarrow (c_1, [\![v_1]\!]); \mathsf{inc}(f\ a) && \text{(By Items 1 and 2)} \\
&= \mathsf{step}^{c}(\mathsf{step}^{c_1}(\mathsf{inc}((\lambda x.[\![e']\!])\ [\![v_1]\!]))) \\
&= \mathsf{step}^{c+c_1+1^\bullet}([\![e']\!]([\![v_1]\!])) \\
&= \mathsf{step}^{c+c_1+1^\bullet}([\![e'[v_1/x]]\!]) && \text{(By } \textbf{(5.2.1∗9)}\text{)} \\
&= \mathsf{step}^{c+c_1+1^\bullet}(c_2, [\![v]\!]) && \text{(By Item 3)} \\
&= (c + c_1 + 1^\bullet + c_2, [\![v]\!]) && \text{(By definition of } \mathsf{step}\text{)}
\end{aligned}$$

But this is what we needed to show.

5.2.4. Computational adequacy.

(5.2.4∗1) Theorem 5.2.3∗1 states that ground operational equivalences are denotational equalities, which we proved by exploiting the fact that operational equivalences (*i.e.* propositions of the form $e \Downarrow^c v$) can be characterized inductively. On the other hand, it is not as straightforward to establish *adequacy* in the converse direction since an equation $[\![e]\!] = (c, [\![v]\!])$ has no analogous mapping-out property; induction on the derivation of terms also fails.

(5.2.4∗2) Explain the problem one encounters when trying to prove the statement "for all $e, v : 2$, if $[\![e]\!] = (c, [\![v]\!])$, then $e \Downarrow^c v$." directly by induction on the derivation of $e : 2$.

(5.2.4∗3) One of the first lessons a PL student learns is that almost all nontrivial properties about higher-order languages (programming language with function types) cannot be proved by induction on the derivation of terms. A major breakthrough was made by Tait, who had the profound insight that nontrivial semantic properties may be established by generalizing the property to a *family* of type-indexed properties $\mathcal{P}_A$, defined so that property at the function type $A \to B$ is the preservation of the property at A and B. This construction/proof strategy is commonly called a *logical relations* or *Tait computability* argument.

(5.2.4∗4) In our current context, it was Plotkin who used a binary logical relations between syntax and semantics to establish computational adequacy in the sense of **(5.1∗10)**. We follow *op. cit.* and prove cost-sensitive generalizations of the classic computational adequacy property by means of a logical relations construction.

(5.2.4∗5) The *formal approximation relations* are a family of relations $\lhd_A \subseteq [\![A]\!] \times \mathsf{tm}(A)$ defined by recursion on the structure of types:

$$\begin{aligned}
&e \lhd_2 e' \iff e\ \mathsf{val} \land [\![e']\!] = e \\
&e \lhd_{A \to B} e' \iff \exists[x : A \vdash e_2 : B]\ (e' = \lambda x.e_2') \land (\forall[e_1 \lhd_A e_1']\ e\ e_1 \lhd_B^{\Downarrow} e_2'[e_1'/x])
\end{aligned}$$

In the above we write $R^{\Downarrow}$ for the lift of a relation $R \subseteq [\![A]\!] \times \mathsf{tm}(A)$ to a relation $R^{\Downarrow} \subseteq [\![A]\!] \times \mathbb{C} \times \mathsf{tm}(A)$:

$$e\ R^{\Downarrow}\ e' = \forall[v : [\![A]\!], c : \mathbb{C}]\ e = (c, v) \to \exists[v' : \mathsf{tm}(A)]\ (e' \Downarrow^c v') \land (v \lhd_A v')$$

(5.2.4∗6) Unfolding definition, we have that the proposition $[\![e]\!] \lhd_2 e$ is just the cost-sensitive adequacy property we want to establish. Thus it suffices to show that all terms $e : A$ have the property that $[\![e]\!] \lhd_A e$ holds. To state this precisely, we first lift the formal approximation relations to contexts.

📖 **(5.2.4∗7)** We may define a family of relations $\lhd_\Gamma \subseteq [\![\Gamma]\!] \times \mathsf{Sub}(\cdot, \Gamma)$ indexed in a context Γ between the corresponding semantic contexts and closing substitutions:

$$\begin{aligned} &\cdot \lhd_{\cdot} * \iff \top \\ &(\gamma, v) \lhd_{\Gamma, x:A} s, v'/x \iff (\gamma \lhd_\Gamma s) \wedge (v \lhd_A v') \end{aligned}$$

□ **(5.2.4∗8)** Formal approximation relations are closed under *head expansion*: if $e \lhd^{\Downarrow}_A e''$ and $e' \mapsto e''$, then $\mathsf{inc}(e) \lhd^{\Downarrow}_A e'$.

■ **(5.2.4∗9)** This follows from the fact that $e'' \Downarrow^c v$ and $e' \mapsto e''$ implies $e' \Downarrow^{1^\bullet + c} v$.

□ **(5.2.4∗10)** If $e \lhd^{\Downarrow}_A e''$ and $e' \xrightarrow{c}^* e''$, then $\mathsf{step}^c(e) \lhd^{\Downarrow}_A e'$.

■ **(5.2.4∗11)** By induction on the derivation of $e' \xrightarrow{c}^* e''$ and **(5.2.4∗8)**.

5.2.5. Fundamental lemma of logical relations.

(5.2.5∗1) We write $\Gamma \vdash e \lhd_A e'$ when for all $\gamma \lhd_\Gamma s$, we have that $e(\gamma) \lhd^{\Downarrow}_A e'[s]$.

(5.2.5∗2) We aim to prove the *fundamental lemma of logical relations*: $\Gamma \vdash [\![e]\!] \lhd_A e$ for all terms $\Gamma \vdash e : A$.

□ **(5.2.5∗3)** We have that $[\![\mathsf{yes}]\!] \lhd_2 \mathsf{yes}$ and $[\![\mathsf{no}]\!] \lhd_2 \mathsf{no}$.

■ **(5.2.5∗4)** By definition of values **(5.2.1∗5)** and formal approximation at 2 **(5.2.4∗5)**.

(5.2.5∗5) Given a map $f : A \times B \to C$, we write $\tilde{f} : A \to C^B$ for its exponential transpose.

(5.2.5∗6) Given maps $f : G \to \mathbb{C} \times B^A$ and $g : G \to \mathbb{C} \times A$, we write $\mathsf{ev}(f, g) : G \to \mathbb{C} \times B$ for the *monadic* evaluation map:

$$\mathsf{ev}(f, g)(x) = f' \leftarrow f(x); a \leftarrow g(x); \mathsf{inc}(f'(a))$$

□ **(5.2.5∗7)** If $\Gamma, x : A \vdash e \lhd_B e'$, then $\Gamma \vdash \tilde{e} \lhd_{A \to B} \lambda x.e'$.

■ **(5.2.5∗8)** Fixing a compatible pair $\gamma \lhd_\Gamma s$, we need to show that $\tilde{e}(\gamma) \lhd_{A \to B} (\lambda x.e')[s]$. Unfolding the definition of formal approximations, we further suppose $v \lhd_A v'$ and aim to to show $\tilde{e}(\gamma)(v) \lhd^{\Downarrow}_B e'[s][v'/x]$, which is equivalent to showing $v \lhd_A v'$ and aim to to show $e(\gamma, v) \lhd^{\Downarrow}_B e'[s, v'/x]$. Observing that $(\gamma, v) \lhd_{\Gamma, x:A} s, v'/x$, the result then follows from the assumption that $\Gamma, x : A \vdash e \lhd_B e'$ holds.

□ **(5.2.5∗9)** If $\Gamma \vdash e \lhd_{A \to B} e'$ and $\Gamma \vdash e_1 \lhd_A e_1'$, then $\Gamma \vdash \mathsf{ev}(e, e_1) \lhd_B e'\ e_1'$.

■ **(5.2.5∗10)** Fixing $\gamma \lhd_\Gamma s$, we must show $\mathsf{ev}(e, e_1)(\gamma) \lhd^{\Downarrow}_B (e'\ e_1')[s]$, which means to show $\mathsf{ev}(e, e_1)(\gamma) \lhd^{\Downarrow}_B e'[s]\ e_1'[s]$. By the premises we have the following:

1. $e(\gamma) \lhd^{\Downarrow}_{A \to B} e'[s]$.
2. $e_1(\gamma) \lhd^{\Downarrow}_A e_1'[s]$.

Writing $(c_1, f) = e(\gamma)$ and $(c_2, a) = e_1(\gamma)$, this means we have $e'[s] \Downarrow^{c_1} f'$ and $e_1'[s] \Downarrow^{c_2} a'$ for some f' and a' such that $f \lhd_{A \to B} f'$ and $a \lhd_A a'$. Because f' is a value, it must take the form $\lambda x.e'$ for some e'. By the definition of formal approximation, we then have that $f(a) \lhd_B^{\Downarrow} e'[a'/x]$. Since we have $e'[s]\ e_1'[s] \xrightarrow{c_1}{}^* (\lambda x.e')\ (e_1'[s]) \xrightarrow{c_2}{}^* (\lambda x.e')\ a' \mapsto e'[a'/x]$, the result then follows by **(5.2.4∗10)** and computing in the denotational semantics:

$$\begin{aligned}
\mathsf{ev}(e, e_1)(\gamma) &= f \leftarrow e(\gamma); a \leftarrow e_1(\gamma); \mathsf{inc}(f\ a) \\
&= f \leftarrow (c_1, f); a \leftarrow (c_2, a); \mathsf{inc}(f\ a) \\
&= \mathsf{step}^{c_1 + c_2}(\mathsf{inc}(f\ a)) \\
&= \mathsf{step}^{c_1 + c_2 + 1^{\bullet}}(f\ a)
\end{aligned}$$

☐ **(5.2.5∗11)** The fundamental lemma of logical relations **(5.2.5∗2)** holds.

■ **(5.2.5∗12)** By **(5.2.5∗3)**, **(5.2.5∗7)** and **(5.2.5∗9)**.

︾ **(5.2.5∗13)** Given a closed program $e : 2$, we have that $[\![e]\!] = (c, [\![v]\!])$ if and only if $e \Downarrow^c v$.

■ **(5.2.5∗14)** By Theorem 5.2.3∗1 and **(5.2.5∗11)**.

☐ **(5.2.5∗15)** Cost-sensitive computational adequacy *restricts* to a classic computational adequacy theorem in the functional phase: given a closed program $e : 2$, we have that $[\![e]\!] = (- : 1, [\![v]\!])$ if and only if $e \Downarrow v$.

■ **(5.2.5∗16)** In the functional phase, the cost monoid $\mathbb{C}$ restricts to a point, and so the proposition $e \Downarrow^c v$ is equivalent to $\exists[c : \mathbb{C}]\ e \Downarrow^c v$.

5.3. COST MODELS

(5.3∗1) Orthogonal to the distinction between denotational and operational cost semantics, cost analysis of programs may be conducted relative to different *cost models*. Generally speaking cost models are divided into whether they operate at a language level or at an algorithmic/problem level. For instance **(4.3.2∗2)** and **(4.3.2∗3)** exhibits denotational cost analyses in which the cost model varies with respect to different problems. On the other hand, Sections 5.2.1 and 5.2.2 illustrate language-level cost models in the denotational and operational style, respectively, in which the cost of programs are given uniformly in a predetermined fashion. Thus we have the following classification of approaches to cost analysis in **calf**:

	Algorithmic-level cost model	language-level cost model
Denotational	Euclid's algorithm **(4.3.2∗2)**, Sorting **(4.3.2∗3)**	**STLC** Section 5.2.1
Operational		**STLC** Section 5.2.2

(5.3∗2) The role of cost-sensitive computational adequacy is to relate the first row of **(5.3∗1)** to the second row. For example, a **calf** cost bound on any function f in the image of the denotational cost semantics of the **STLC** is also an *operational* cost bound for some **STLC** program $\overline{f}$ by appealing to cost-sensitive computational adequacy.

CHAPTER 6

Domain theory

(6∗1) The last few chapters of this dissertation aim to generalize the results of Chapter 5 to Plotkin's **PCF**, a language that features general recursion. Recall that the call-by-push-value decomposition of **PCF** is defined as the following inductive family of terms:

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{nat}} \qquad \frac{\Gamma \vdash v : \mathsf{nat}}{\Gamma \vdash \mathsf{suc}(v) : \mathsf{nat}}$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \qquad \Gamma \vdash e_0 : X \qquad \Gamma, z : \mathsf{nat} \vdash e_1 : X}{\Gamma \vdash \mathsf{ifz}(e, e_0, z.e_1) : X} \qquad \frac{\Gamma, x : A \vdash e : X}{\Gamma \vdash \lambda x.e : A \to X}$$

$$\frac{\Gamma \vdash e : A \to X \qquad \Gamma \vdash e_1 : A}{\Gamma \vdash e\ e_1 : X} \qquad \frac{\Gamma, x : \mathsf{U}X \vdash e : X}{\Gamma \vdash \mathsf{fix}(x.e) : X} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{ret}(a) : \mathsf{F}A}$$

$$\frac{\Gamma \vdash e : \mathsf{F}A \qquad \Gamma, a : A \vdash e_1 : X}{\Gamma \vdash \mathsf{bind}(e, a.e_1) : X}$$

There are two purposes of considering the call-by-push-value version of **PCF**. First, as observed in **(4.1.3∗9)**, separating values and computations at the level of types allows us to smoothly integrate cost structure as an abstract computational effect. Second, since both call-by-value and call-by-name **PCF** have canonical decompositions in call-by-push-value **PCF**, by framing our work around the general case we easily obtain the corresponding results for call-by-value and call-by-name by composing with the respective decompositions.

(6∗2) A general recursive function in **PCF** permits function definitions such as the following:

$f : \mathsf{Fnat} \to \mathsf{Fnat}$
$f = \mathsf{fix}(\lambda f, x.\mathsf{bind}(f\ x; \lambda n.\mathsf{ret}(n+1)))$

which satisfies the following equation:

$$f(x) = \mathsf{bind}(f\ x; \lambda n.\mathsf{ret}(n+1)))$$

The function f cannot shown to be well-defined by means of structural recursion. Instead, functions of **PCF** are defined by means of a *fixed-point semantics* in which recursive definitions are interpreted as (least) fixed-points.

(6∗3) Construed as a function $\mathbb{N} \to \mathbb{N}$ in **Set**, f clearly cannot possess any fixed-points. Intuitively one should think about the meaning of $f(x)$ as a *divergent* or *undefined* value that infects every subsequent computation, rendering $f(x)$ and $\mathsf{bind}(f\ x; \lambda n.\mathsf{ret}(n+1)))$ to be equally undefined values. Mathematically we accomplish this by an operation called *lifting* that adjoins a new value $\perp$ to $\mathbb{N}$ and assigning the meaning of f to be the totally undefined map determined by $\perp$, rendering $\perp$ a fixed-point of f.

(6∗4) In general merely extending sets with a distinguished element $\perp$ is not enough to give a proper account of the fixed-point semantics of programs. The semantic picture is that one also needs to enrich sets with an order relation called the *information order* encoding the amount of definite information is conveyed by any element. This suggests that our semantic domain is a category of posets in which functions are monotone with respect to the information order. In fact when working with posets with strong enough completeness properties, monotonicity alone is enough to guarantee the existence of fixed-points [92]. However, to obtain a tight correspondence between the fixed-point and the operational semantics, one commonly works with *continuous* maps that in addition preserve the existing joins in a chosen class of posets. From an information-theoretic perspective, this can be understood as requiring that the output of a function f on a consistent gluing of elements $\bigvee D$ can be determined as a consistent gluing of pieces of the output $\bigvee f(D)$.

(6∗5) The study of the order-theoretic structure of recursive functions and recursive types is called *domain theory*. In Section 6.1 we outline the relevant domain theory needed for our results; the reader is referred to Scott and Strachey's seminal work for the origins of domain theory and denotational semantics [107, 105, 106] and Abramsky and Jung [2] and Streicher [123] for textbook accounts.

6.1. DOMAIN THEORY

(6.1∗1) A poset D is closed under ω-joins when for all ascending chains $\mathbb{N} \to D$ is equipped with a least upper bound. An ω-cpo is a poset closed under all ω-joins.

(6.1∗2) Observe that a poset that is orthogonal **(3.5∗2)** to the *figure shape* $\{0 \sqsubseteq 1 \ldots\} \hookrightarrow \{0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq \infty\}$ is equipped with an upper bound for every ω-directed set:

$$\begin{tikzcd}
\{0 \sqsubseteq 1 \ldots\} \arrow[r, tail] \arrow[d] & \{0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq \infty\} \arrow[dl, dotted] \\
D &
\end{tikzcd}$$

Thus a such poset D is an ω-cpo if and only if every down set $\downarrow(d)$ is also orthogonal to $\{0 \sqsubseteq 1 \ldots\} \hookrightarrow \{0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq \infty\}$.

(6.1∗3) A function $D \to E$ between ω-cpo's *continuous* when it preserves ω-joins. We write ω**CPO** for the category of ω-cpo's and monotone and continuous functions.

(6.1∗4) A *pointed* ω-cpo or ω-cppo is an ω-cpo equipped with a least element, usually denoted as $\perp$.

(6.1∗5) A continuous function $D \to E$ is *strict* when it preserves least elements. We write $\omega\mathbf{CPPO}$ for the category of pointed ω-cpo's and strict maps.

(6.1∗6) The *Kleene chain* $\mathsf{kl}_f : \mathbb{N} \to D$ associated to an endomap $f : D \to D$ of pointed ω-cpo's is defined to be the finite n-fold self-composition of f applied to the least element: $\mathsf{kl}_f(n) = f^{(n)}\bot$.

(6.1∗7) The main purpose of considering posets closed under ω-joins is *Kleene's fixed-point theorem*: every endomap $f : D \to D$ of pointed ω-cpo's possesses a least fixed-point, defined as the ω-directed join of its associated Kleene chain.

6.1.1. Constructions on domains.

(6.1.1∗1) The *free ω-cpo* on a set X equips X with the *discrete* or *flat* ordering in which $x \sqsubseteq y$ if and only if $x = y$. We sometimes write $\mathsf{Disc}(X)$ for the free ω-cpo on X; it has the following universal property with respect to every ω-cpo D:

$$\begin{tikzcd}
X \arrow[r] \arrow[d] & \mathsf{Disc}(X) \arrow[dl, dotted] \\
D &
\end{tikzcd}$$

In other words, *every* (not necessarily continuous) function $f : X \to D$ extends to a unique continuous function $\overline{f} : \mathsf{Disc}(X) \to D$. This construction extends to an adjunction $\omega\mathbf{CPO} \underset{\longrightarrow}{\overset{\longleftarrow}{\perp}} \mathbf{Set}$ in which the left adjoint Disc sends a set X to the discrete ω-cpo X and the right adjoint sends an ω-cpo to it underlying set.

(6.1.1∗2) The *free ω-cppo* on an ω-cppo has the underlying set $D_\bot = \{\bot\} \sqcup D$ extending D with a new distinguished element $\bot$. The order relation on $D_\bot$ is defined by $x \sqsubseteq y$ if and only if $x = \bot$ or $x \sqsubseteq y$ in D. This construction has the following universal property:

$$\begin{tikzcd}
X \arrow[r, tail] \arrow[d] & X_\bot \arrow[dl, dotted] \\
D &
\end{tikzcd}$$

in which a continuous map out of X extends to a *strict* (bottom-preserving) continuous map on $X_\bot$. The free ω-cppo construction extends to a functor $\bot : \omega\mathbf{CPO} \to \omega\mathbf{CPPO}$ left adjoint to the forgeful functor $\omega\mathbf{CPPO} \to \omega\mathbf{CPO}$.

(6.1.1∗3) A domain may be equivalently defined as an *algebra* (see Section 3.2.4) for the lift monad $\mathbb{L}$ determined by the free-forgetful adjunction $\omega\mathbf{CPPO} \underset{\longrightarrow}{\overset{\longleftarrow}{\perp}} \omega\mathbf{CPO}$.

(6.1.1∗4) The category of pointed ω-cppo's $\omega\mathbf{CPPO}$ is equivalent to the category of lift algebras on $\omega\mathbf{CPO}$.

(6.1.1∗5) The category of ω-cpo's is Cartesian closed, with products and exponential defined pointwise. The adjunction $\omega\mathbf{CPPO} \underset{\longrightarrow}{\overset{\longleftarrow}{\perp}} \omega\mathbf{CPO}$ furnishes a model of call-by-push-value **PCF** in which value types are interpreted as ω-cpo's and computation types are interpreted as lift algebras. The fixed-point operator $\mathsf{fix} : (X \to X) \to X$ is interpreted as the least fixed-point operator by means of **(6.1∗7)**.

6.2. CONSTRUCTIVE DOMAIN THEORY

(6.2∗1) The domain theory outlined Section 6.1 is sufficient to give a denotational semantics of ordinary **PCF**. To account for cost structure, we would like to work in a topos model of the FC phase distinction. In other words, instead of defining ω-cpos relative to **Set**, we work internally to some topos $\mathscr{E}$ equipped with a phase distinction ϕ.

(6.2∗2) As explained in Section 3.2, the internal language of a topos is an intuitionistic type theory. On the other hand, the definition of the free ω-cpo on a set given in **(6.1.1∗2)** necessitates the use of classical principles that are not constructively valid. In particular, the proposition that $X_\perp$ is closed under ω-joins is equivalent to the *limited principle of omniscience* [31, Proposition 3.4.1]:

(LPO) Every binary sequence $\mathbb{N} \to 2$ either contains a 1 or not.

For the maximum applicable of results, when working axiomatically in the internal language of a topos it is crucial that we do not rely on non-constructive principles such as LPO.

(6.2∗3) In a topos, the counterpart to the lifting operation of **(6.1.1∗2)** is given by the *partial map classifier* (defined in **(6.2.1∗6)**), giving rise to a constructive version of the theory outlined in Section 6.1. In Section 6.2.1 we present the necessary notions of this constructive domain theory from de Jong [31]. Although we will follow *op. cit.* and work in terms of *directed complete partial orders* (dcpos), the results of this thesis do not rely on the full generality of dcpos.

6.2.1. Constructive domains.

(6.2.1∗1) This section should be understood in the internal language of a topos. The external meaning of internal definitions and theorems may be unfolded by means of the Kripke-Joyal semantics as explained in Section 3.2.2.

📖 **(6.2.1∗2)** A *directed family* in a poset D is a family $\alpha : I \to D$ such that every pair of elements $\alpha_i, \alpha_j : D$ has an upper bound.

📖 **(6.2.1∗3)** A *dcpo* is a poset closed under all directed joins. A *pointed dcpo* or *dcppo* is a dcpo equipped with a least element. A morphism of dcpos is a function $D \to D$ preserving all directed joins.

☐ **(6.2.1∗4)** Every dcpo is an ω-cpo.

⌄ **(6.2.1∗5)** Every pointed dcpo endomap $X \to X$ possesses a unique fixed-point.

📖 **(6.2.1∗6)** In contrast to ω-cpos, lifting of dcpos is defined in terms of the *partial element classifier*. The action of lifting on points is defined as follows. $\mathsf{L}(A) = \Sigma_{\phi:\Omega}.\phi \to A$. Partial elements $u, v : \mathsf{L}A$ are ordered in a pointwise fashion. Writing $u{\downarrow} : \Omega$ for the *termination support* of u, we define $u \sqsubseteq v$ to hold if and only if $u{\downarrow}$ implies $v{\downarrow}$ and whenever $u{\downarrow}$ holds, $u \sqsubseteq v$ in the dcpo A.

🔨 **(6.2.1∗7)** The lifting functor L has the structure of a monad $\mathbb{L} = (\mathsf{L}, \eta, \mu)$:

$$\eta_A : A \to \mathsf{L}A$$
$$\eta_A(a) = (\top, \lambda - .a)$$

$$\mu_A : \mathsf{L}(\mathsf{L}A) \to \mathsf{L}A$$
$$\mu_A(\phi, f) = (\exists[u : \phi]\ \psi(u), a)$$

In the above, we write $\psi : \phi \to \Omega$ for the map $\downarrow \circ f$ and $a : \exists[u : \phi]\ \psi(u) \to A$ for the partial element determined by $\pi_2 f$.

(6.2.1∗8) Observe that the subobject classifier $\Omega = \mathsf{L}1$ is always an internal dcpo, with directed-joins computed by means of the ambient topos join $\exists$.

6.2.2. Closure properties of domains.

(6.2.2∗1) For any universe $\mathcal{U}$ in $\mathscr{E}$ we have *internal* categories of $\mathcal{U}$-small dcpos and pointed dcpos. However since internal categories are somewhat unwieldy, we will work with ordinary categories of domains (*i.e.* a category internal to **Set**) given by externalizing internal categories. Fixing $\mathscr{E} = \widehat{I}$ to be a presheaf topos, we have a category $\mathscr{D}$ of internal dcpos fibred over $\widehat{I}$. The fibre category over the terminal object of $\widehat{I}$ obtains us an ordinary category $\mathscr{C}$ consisting of (small[1]) internal dcpos.

☐ **(6.2.2∗2)** $\mathscr{C}$ is Cartesian closed with products and exponentials computed pointwise [31].

(6.2.2∗3) To get a category of internal dcppos, we observe that the internal lifting monad restricts to an ordinary monad $\mathscr{C} \to \mathscr{C}$. Define $\mathscr{C}^{\mathbb{L}}$ to be category of dcppos, *i.e.* the category of lift algebras over $\mathscr{C}$.

(6.2.2∗4) Thus similar to the situation of ω-cpos, we again have an adjunction $\mathscr{C}^{\mathbb{L}} \underset{\longrightarrow}{\overset{\longleftarrow}{\perp}} \mathscr{C}$ suitable to give the denotational semantics of **PCF**. Moreover, by instantiating the construction at the presheaf topos $\widehat{\mathbb{I}}$, we obtain an account of the denotational *cost* semantics of **PCF**.

☐ **(6.2.2∗5)** The intermediate proposition $\mathsf{u} : \widehat{\mathbb{I}}$ is a dcpo.

■ **(6.2.2∗6)** Because every directed family in a proposition ϕ determines a proof $u : \phi$, from which the result follows since $\phi = 1$ is clearly a dcpo.

☐ **(6.2.2∗7)** For every dcpo A, $\bullet A$ is a dcpo.

■ **(6.2.2∗8)** This follows from the cocompleteness of dcpos.

(6.2.2∗9) Thus we may interpret the (partial) cost effect $\mathsf{F}A$ by lifting the interpretation of A paired with a sealed discrete dcpo $\mathbb{C}$ serving as the cost monoid. The idea is that such a

[1]Externally, a type-theoretic universe $\mathcal{U}$ in $\mathscr{E}$ is given by the Hofmann-Streicher lifting [60] of an ambient Grothendiekc universe U. A $\mathcal{U}$-small internal category then externalizes to an ordinary U-small category.

cost-sensitive model restricts to an ordinary model of the pure functional semantics of **PCF** in the slice category $\mathscr{C}/\mathsf{u}$.

(6.2.2∗10) Currently the observations of **(6.2.2∗9)** are of limited utility because they are external statements about $\mathscr{C}$. To fully reap the benefits of the phase distinction as a tool for reasoning about cost-sensitive programs, we will embed $\mathscr{C}$ in a model of synthetic domain theory (see Chapter 7), integrating domain-theoretic maps with the full logical facilities provided by the internal type theory of a topos.

6.3. TOPOLOGICAL VIEW OF DOMAINS

(6.3∗1) Although the order-theoretic presentation of domains may seem somewhat esoteric from the point of view of mainstream mathematics, a domain (more precisely a dcpo) can be also seen as a kind of *space*, and many concepts of domain theory may be motivated from a topological point of view. We will outline this view of domains as spaces, which will serve to provide some geometric intuition when we move on to synthetic domain theory in Chapter 7.

(6.3∗2) Every dcpo D may be equipped with a topology called the *Scott topology*. The open sets of this topology are given by subsets of D that are both upwards-closed (up sets) and *inaccessible by directed join*. This latter condition means that for every directed subset $S \subseteq X$, if $\bigvee S$ is contained in an open set U, we have that there exists $s \in S$ such that already $s \in U$. In other words, one cannot start with a directed set disjoint from U and reach U via directed joins.

(6.3∗3) The Scott topology on a discrete dcpo is the discrete topology.

(6.3∗4) Computationally, we think of an open subset as an *observable* or *computational* predicate. The fact that observations are required to be upwards-closed corresponds to the following property: computationally speaking, once an element has been *apprehended*, *i.e.* deemed to belong to an observation class, any further (consistent) information about the element should not negate its apprehension. For example, any observation that apprehends the undefined element $\perp$ must also apprehend every element (of the domain).

(6.3∗5) Moreover, the inaccessibility of open sets by directed joins evinces the *finite character* of observations. Taking the example of an observation over functions $\Phi(f)$, whenever a recursive function $\mathsf{fix}(f)$ is apprehended by Φ, so must a finite prefix $f^{(n)}$ of $\mathsf{fix}(f)$ have been apprehended as well.

(6.3∗6) Recall the *Sierpiński space* Σ from **(3.2.2∗9)**, whose open sets are given by the up sets on the interval $\mathbb{I} = \{0 \sqsubseteq 1\}$. In a classical metatheory, Σ has the following universal property: it is the *open subset classifier*. More precisely, this means that every open $U \hookrightarrow X$ of a space X arises as the pullback of the *generic open* $\top : 1 \to \Sigma$ determined by $\{1\} \in \mathcal{O}(\Sigma)$:

$$\begin{tikzcd}
U \arrow[r] \arrow[d, hook] \arrow[dr, phantom, "\lrcorner", very near start] & 1 \arrow[d, "\top"] \\
X \arrow[r] & \Sigma
\end{tikzcd}$$

The characteristic map $X \to \Sigma$ is defined as follows.

$$x \mapsto \begin{cases} 1 & \text{if } x \in U \\ 0 & \text{o.w.} \end{cases}$$

Conversely, the open subset associated to a map $f : X \to \Sigma$ is given by $f^{-1}(1)$.

(6.3∗7) In the internal domain theory of an arbitrary (non boolean) topos, the Sierpiński space Σ is defined to be the ind-completion/ideal-completion/free dcpo on $\{0 \sqsubseteq 1\}$, *i.e.* the subobject classifier Ω (in a classical metatheory the ind-completion is identity on $\{0 \sqsubseteq 1\}$). Thus constructively, we have that Scott-open subsets of a dcpo D are in unique correspondence with continuous predicates $X \to \Sigma$. These subsets play an important role in Chapter 7 as they are used to define finite coproducts in the category of internal dcpos **(6.2.2∗1)**.

(6.3∗8) One may associate a preorder relation called the *specialization preorder* to every topological space X: we define $x \sqsubseteq y$ if and only if every open set containing x also contains y.

(6.3∗9) In view of **(6.3∗7)**, we have that the specialization preorder on a dcpo D is such that $x \sqsubseteq y$ if and only if $f(x)$ implies $f(y)$ for every continuous $f : D \to \Sigma$. One may verify that this coincides with the original order on D. In synthetic domain theory we call the specialization order on a type its *intrinsic order*.

(6.3∗10) Use **(6.3∗9)** to show that a function between dcpos $D \to E$ is continuous (in the domain-theoretic sense) if and only if it is continuous (in the topological sense with respect to the Scott topology).

CHAPTER 7

Synthetic domain theory

(7∗1) The goal of this chapter is to integrate the type theory **calf** of Chapter 4 with the domain-theoretic structure discussed in Chapter 6. Our tool of choice to produce semantic models of such an integration is *synthetic domain theory* (SDT) [62].

(7∗2) Synthetic domain theory as a field started when Dana Scott conjectured that one ought to be able to reason about domains as if they were just *special* sets provided that one employs a constructive ambient metalanguage. In technical terms, this metalanguage can be construed as the internal languages of a topos, *i.e.* an extensional dependent type theory. In general it is difficult to reconcile the domain-theoretic structure with the rich logical structure of dependent type theory, so the quest for SDT was in essence about constructing full subcategories of topoi that supported domain-theoretic constructions. Fullness is a critical property — it means that *every* map definable in type-theoretic language is a domain morphism, which absolves one from checking onerous side conditions (namely monotonicity and continuity of maps) when working internally.

(7∗3) There are two well-known ways to obtain models of synthetic domain theory: one based on realizability (Hyland [62], Phoa [94], Reus [98]) and one based on sheaf topoi (Rosolini [103], Fiore and Rosolini [39], Fiore and Plotkin [38], Matache *et al.* [82]). In this dissertation we will employ *relative* sheaf models of SDT based on the work of Sterling and Harper. The rough idea is that given a basic domain-theoretic site $\mathscr{C}$, the (pre)sheaf topos on $\mathscr{C}$ contains a reflective subcategory of *synthetic predomains* so that the image of every concrete predomain in $\mathscr{C}$ under the Yoneda embedding is a synthetic predomain. A relative sheaf model of SDT just means that instead of an ordinary site $\mathscr{C}$, the construction is based on an *internal* site or a site fibred over some base topos $\mathscr{E}$. For the purposes of modeling cost structure, naturally we will define a site internal to $\widehat{\mathbb{I}}$ (in particular, we will employ the category of internal dcpos as defined in **(6.2.2∗1)**).

(7∗4) The structure of this chapter is as follows. In Section 7.2, we define an axiomatization dubbed *basic domain-theoretic category* (BDTC) that serves to identify appropriate domain-theoretic sites for our sheaf model of SDT. In Section 7.3, we introduce the axioms of *cost-sensitive* synthetic domain theory and show that the category of synthetic predomains admit a model of BDTC. In Section 7.4, we substantiate the axioms of cost-sensitive SDT by constructing a model of cost-sensitive SDT from every BDTC following the outline in **(7∗3)**.

7.1. WHAT IS SYNTHETIC ABOUT SYNTHETIC DOMAIN THEORY?

(7.1∗1) While studying the primary literature on synthetic domain theory the reader is likely to encounter a separate subfield of domain theory called *axiomatic domain theory* (ADT) [34], which aims to both find concrete models of domains and give universal characterizations to domain constructions. Although related, these two subfields of domain theory serve distinct purposes. In this section I will attempt to clarify the picture by illustrating the relationship between synthetic and axiomatic domain theory (in the context of this thesis) through the simpler theory of *preorders*.

7.1.1. Synthetic and axiomatic preorder theory.

(7.1.1∗1) The theory of preorders is not only ideal for illustration purposes but also relevant to the subject of cost analysis, as demonstrated in the work of Grodin et al. on extending the **calf** type theory to account for *higher-order, effectful* programs. Briefly *op. cit.* argue that instead of artificially limiting a cost bound to have the form $A \to \mathbb{C}$ for some monoid $\mathbb{C}$, one should consider general *program inequalities* $e \sqsubseteq e'$, which means that programs should be modeled in a preorder-enriched category with sufficiently nice closure properties to accommodate the existing type connectives of **calf**.

Now a clear candidate for such a category is **Preord**, the category of preorders and monotone maps. However, it is difficult to construct models of dependent type theories out of categories of order-like objects (preorders, posets, categories) because they are not locally Cartesian closed.[1]

(7.1.1∗2) The approach of Grodin et al. [46] (and indeed of synthetic domain theory) is to start, at least conceptually, from the opposite end: isolate within type theory a class of types that behave like preorders, which one may call *synthetic preorders*. The benefit of such a synthetic theory of preorders is that one has a unified language to reason about both general mathematical objects and these distinguished objects equipped with an intrinsic preorder relation. Moreover, one may arrange the semantics of the theory so that *every* internal type-theoretic construction respects the intrinsic preorder (in the case of synthetic domain theory this corresponds to constructions being both monotone *and* continuous).

(7.1.1∗3) A typical (perhaps even defining) feature of synthetic formulations of classical theories such as preorders/domains/smooth spaces is that one aims to find a special object in an ambient type theory or intuitionistic set theory such that the development of the theory can proceed by assuming a few axioms on the distinguished object. In the case of preorders, the *interval* preorder $\mathbb{I} = \{0 < 1\}$ plays the role of the distinguished object.

There are two ways to understand how the interval can be used to derive a theory of synthetic preorders. First, observe that given a preorder P, ordered pairs in P are in one-to-one correspondence with monotone maps $\mathbb{I} \to P$, which we may visualize as *paths* in P. Thus one may express the preorder relation in terms of paths: $x \sqsubseteq_P y \iff \exists[\alpha : \mathbb{I} \to P]\ \partial(\alpha) = (x, y)$, where $\partial(\alpha) = (\alpha\ 0, \alpha\ 1)$ is the *boundary* of the path α.

Alternatively, we may use $\mathbb{I}$ in the dual role as the classifier of upwards closed subsets (*cf.* Σ as the classifier of Scott-open subsets as in **(6.3∗6)**). More precisely, we have that maps $P \to \mathbb{I}$

[1] In particular the pullback functor $f^* : \mathbf{Preord}/P \to \mathbf{Preord}/Q$ does not preserve all colimits and so cannot have a right adjoint $\Pi_f : \mathbf{Preord}/Q \to \mathbf{Preord}/P$, which is used to give the categorical semantics of dependent product types; for details see Hazratpour [55, p. 3].

are in one-to-one correspondence with upwards closed subsets of P. As in the topology view of domains in Section 6.3, there is an analogous *intrinsic/specialization preorder* on P defined by $x \sqsubseteq^{\circ}_{P} y \overset{\text{def}}{=} \forall[f : P \to \mathbb{I}]\ f\ x \sqsubseteq f\ y$, which coincides with the original preorder on P.

(7.1.1∗4) The point of thinking about preorders in terms of maps into/out of the interval object $\mathbb{I}$ is that the definitions of **(7.1.1∗3)** can be used to induce a preorder relation on general types that are not *a priori* preorders, thus furnishing a convenient mechanism for axiomatizing preorders in type theory. More precisely, the synthetic preorder theory of Grodin et al. [46] is entirely derived from the existence of a bipointed object $0, 1 : \mathbb{I}$ satisfying an axiom called *Phoa's principle* for preorders whose domain-theoretic analog we explain in **(7.2.2∗5)**. In *op. cit.* the synthetic preorder is defined in terms of paths[2] in order to ensure that preorders become discrete in the functional phase, a property achieved by requiring $\mathbb{I}$ to be sealed (recall from Section 3.3 that this means $\P \to (\mathbb{I} \cong 1)$). Internal to the type theory, every map $f : P \to Q$ is monotone with respect to the synthetic preorder, which is evident since from a path $\mathbb{I} \to P$ one obtains the required path $\mathbb{I} \to Q$ by postcomposing with f.

(7.1.1∗5) As observed in **(7.1.1∗1)**, one cannot directly interpret the axioms of synthetic preorder theory into **Preord** because the latter is not a model of type theory. A well-known way to make poorly structured categories such as **Preord** into models of type theories is by means of the Yoneda embedding.

Note that $\widehat{\mathbf{Preord}}$ *would* be a model of type theory if we were able to take presheaves on **Preord**, which is unfortunately not a small category. To accommodate the fact that the semantic category is not necessarily small, recall from **(3.1.4∗5)** that a full subcategory $\mathscr{D} \hookrightarrow \mathscr{C}$ is *dense* when the *nerve*/restricted Yoneda embedding $N : \mathscr{C} \to [\mathscr{D}^{\text{op}}, \mathbf{Set}]$ is fully faithful. The idea is that we want a small[3] subcategory $\mathscr{D} \hookrightarrow \mathscr{C}$ that "contains" all the basic pieces from which every object in $\mathscr{C}$ can be reconstructed as a canonical colimit.

In the case of preorders the (small) category Δ of nonempty finite ordinals $\{[n] \mid n \geq 0\}$ and monotone maps furnishes a dense subcategory of **Preord**. Indeed, even just the full subcategory $\Delta_{\sqsubseteq 1}$ suffices, since every preorder P is given by a canonical colimit involving the generating set (given by maps $[0] \to P$) and order relation (given by maps $[1] \to P$). The category of presheaves over Δ then provides a model of type theory that in addition admits a synthetic preorder theory in which the interval object is just the image of the actual interval preorder $\{0 < 1\}$ under the nerve $N : \mathbf{Preord} \hookrightarrow \widehat{\Delta}$.

(7.1.1∗6) We may derive a preorder relation on individual types, but what is the structure of such synthetic preorders as a collection? In other words, what are the categorical properties of the (full, sub) category of synthetic preorders? This is important if we want to be able to implement algorithms (such as the ones discussed in Section 4.3) in a natural and ergonomic programming language. The study of such structural properties may well be dubbed *axiomatic preorder theory*, parallel to the relationship between synthetic and axiomatic domain theory. The axiomatic preorder theory of Grodin et al. [46] can be summarized as requiring synthetic preorders

[2]The path relation is not necessarily transitive and pointwise on functions in all models [35, Example 6.8]; to solve this Grodin et al. [46] restrict attention to the so-called *path-transitive* and *boundary-separated* types.

[3]The reason that smallness is desirable is that a lot of nice properties of the category of presheaves such as Cartesian closure only hold when the base category is small. An alternative would be to assume the existence of Grothendieck universes and relativize everything in sight to a Grothendieck universe U. In fact this is the approach we take in **(6.2.2∗1)** since there is no obvious small dense subcategory of the category of dcpos.

to form a Cartesian closed preorder-enriched category that is closed under dependent products and discretely indexed dependent sums. At a high level, the results of *op. cit.* show that every model of synthetic preorders in the sense of **(7.1.1∗4)** supports a full subcategory of synthetic preorders complying with the axiomatic preorder theory outlined above.

7.2. AXIOMATIC DOMAIN THEORY

(7.2∗1) In this section we recall and outline basic notions in *axiomatic domain theory (ADT)*, the domain-theoretic counterpart to the axiomatic preorder theory discussed in **(7.1.1∗6)**. Similar to the axiomatic theory of preorders, the role of ADT is to provide the structural basis for the semantics of typed programming languages that in addition accounts for higher-order/general recursion. A difference in comparison to the situation of Section 7.1.1 is that ADT is used to give the general axiomatic requirements on *both* the base category and the resulting category of synthetic (pre)domains. This kind of extension of models of ADT to models of SDT in which the category of synthetic predomains is again a model of ADT was pioneered in Fiore and Plotkin [38] and adapted in Sterling and Harper [122] to account for the presence of information flow structures. Although in the discussion for preorders we have fixed **Preord** for the base category, it would be reasonable to also develop general extension results along the lines of Fiore and Plotkin [38] by calibrating the axiomatic preorder theory outlined in **(7.1.1∗6)**.

(7.2∗2) Similar to the concrete categories of domains in Chapter 6, a model of ADT consists of a category $\mathscr{C}$ equipped with a lift monad $\mathbb{L} = (\mathsf{L}, \eta_{\mathbb{L}}, \mu_{\mathbb{L}})$ whose algebras can be thought of as domains. The main ingredient in the (relatively basic) axiomatic domain theory used in this thesis is the familiar *limit-colimit coincidence* evinced by the *inductive fixed-point object*, an object $\omega : \mathscr{C}$ that is both an initial L-algebra and a final L-coalgebra. Here *inductive* refers to a property of the fixed-point object used to link the axiomatic development to the ambient *synthetic* domain theory (which we discuss in Section 7.3).

(7.2∗3) For a much fuller and more detailed account of axiomatic domain theory, we refer the reader to Marcelo Fiore's dissertation [34].

7.2.1. Partial maps, dominance, lifting.

(7.2.1∗1) In a category $\mathscr{C}$ with pullbacks, a *partial map* $A \rightharpoonup B$ is a span $A \hookleftarrow D \to B$ consisting of a monomorphism $D \hookrightarrow A$ on which $A \rightharpoonup B$ is defined; we also call $D \hookrightarrow A$ the *termination support* of a partial map.

(7.2.1∗2) To obtain a category of partial maps, the class of monos serving as the domain of definition must be closed under identity and composition. Such a class is called a *dominion* in Rosolini [103].

(7.2.1∗3) Similar to how we built a synthetic preorder theory in Section 2.3 around the interval object $\mathbb{I}$, we will base the development of synthetic domain theory around a fundamental object Σ equipped with two distinguished points $\bot, \top : \Sigma$ called the *dominance* that has the following roles.

1. The dominance Σ classifies the dominion, *i.e.* the domain of definition of every partial map $A \hookleftarrow D \to B$ arises from a unique map $A \to \Sigma$. By analogy to concrete categories of predomains **(6.3∗6)** we also call these subsets *Scott-open* subsets.

2. A map $f : \Sigma \to A$ can be thought of as a *path* in A whose endpoints are determined by the boundary $(f\ \bot, f\ \top)$.

Similar to the situation in **(7.1.1∗3)**, one may induce a *domain-theoretic/information-theoretic* (pre)order on predomains by means of the specialization or path relation relative to Σ, whose definition and relationship we elaborate in Sections 7.3.2 to 7.3.4. Although the somewhat dual roles of the dominance might seem strange at first, we show in **(7.2.2∗4)** that it is semantically natural to classify paths in a type in terms of Σ.

(7.2.1∗4) Given a category $\mathscr{C}$ with finite limits, we fix an object Σ equipped with a map $\top : 1 \to \Sigma$. A Σ-*subset* or Σ-*subobject* is a (necessarily monomorphic) map $U \hookrightarrow X$ obtained as the pullback of $\top$ along some $X \to \Sigma$. We say that $U \hookrightarrow X$ is classified by Σ when there is a unique characteristic map $X \to \Sigma$ making the following a pullback diagram:

$$\begin{tikzcd}
U \arrow[r] \arrow[d, hook] \arrow[dr, phantom, "\lrcorner", very near start] & 1 \arrow[d, "\top"] \\
X \arrow[r] & \Sigma
\end{tikzcd}$$

A Σ-*partial map* is a partial map whose support is a Σ-subset.

(7.2.1∗5) We call $(\Sigma, \top)$ a *dominance* when the following conditions hold [38].

1. Every Σ-subset is classified by Σ and the collection of Σ-subsets form a dominion.

2. The base change functor $\top^* : \mathscr{C}/\Sigma \to \mathscr{C}$ along $\top : 1 \to \Sigma$ sending a family $\begin{matrix} X \\ \downarrow \\ \Sigma \end{matrix}$ to the corresponding Σ-subset has a right adjoint called the *lift structure*.

More explicitly, the second condition states that every Σ-partial map $A \hookleftarrow U \to B$ corresponds to a unique *total* map $A \to \mathsf{L}(B)$. When $\mathscr{C}$ is locally Cartesian closed, we may define the right adjoint $\top_* : \mathscr{C} \to \mathscr{C}/\Sigma$ by sending X to the first projection $\begin{matrix} \Sigma_{\phi:\Sigma}.\phi \to X \\ \downarrow \\ \Sigma \end{matrix}$. The right adjoint gives rise to a functor $\mathsf{L} : \mathscr{C} \to \mathscr{C}$ called the *lift functor* that sends an object X to the dependent sum $\Sigma_{\phi:\Sigma}.\phi \to X$.

(7.2.1∗6) In the internal language of a topos, a dominance is just a collection of propositions $\phi : \Omega$ closed under $\top : \Omega$ and dependent sums: $\phi : \Sigma$ and $f : \phi \to \Sigma$ implies $\Sigma_{\phi:\Sigma}.f : \Sigma$. Observe that $\Sigma_{\phi:\Sigma}.f$ is a proposition because given $(u, p) : \Sigma_{\phi:\Sigma}.f$ and $(u', p') : \Sigma_{\phi:\Sigma}.f$, we have $u =_\phi u'$ since ϕ is a proposition and $p =_{f(u)} p'$ is a proposition since $f(u) =_\Sigma f(u')$ is a proposition. We also write $\phi \angle f$ for the dependent sum $\Sigma_{\phi:\Sigma}.f$.

(7.2.1∗7) For domain-theoretic applications, it is necessary to also assume that the empty sub-domain of any domain is a Σ-subset. Therefore, we also assume that the dominance also classifies subsets of the form $\emptyset \to X$ whenever the ambient category has an initial object $\emptyset$.

(7.2.1∗8) The subobject classifier $\top : 1 \to \Omega$ always determines a dominance.

(7.2.1∗9) In a topos, both the subobject classifier Ω and the collection $\{\bot, \top\}$ with the map determined by the element $\top$ form dominances; the former classifies all subsets by definition, and the latter classifies decidable subsets.

(7.2.1∗10) In a category of (pre)orders, the internal $\mathbb{I} = \{0 \sqsubseteq 1\}$ and the map $1 : 1 \to \mathbb{I}$ form a dominance classifying upwards-closed subsets.

(7.2.1∗11) In a category of domains, the internal $\mathbb{I}$ is a dominance classifying Scott-open subsets.

(7.2.1∗12) In a category of internal domains **(6.2.2∗1)**, the subobject classifier of the ambient topos and the map $\top : 1 \to \Omega$ form a dominance classifying (internally) Scott-open subsets.

(7.2.1∗13) In a topos satisfying countable choice, the set of semi-decidable propositions $\{\phi : \Omega \mid \exists[f : \mathbb{N} \to 2]\ (\phi = (\exists[n : \mathbb{N}]\ f\ n = 1))\}$ forms a dominance called the *Rosolini dominance* classifying the semi-decidable subsets.

(7.2.1∗14) The construction in **(6.2.1∗7)** shows that the lift structure relative to the dominance determined by the subobject classifier has the structure of monad. Observe that this generalizes to any dominance Σ because 1) monad unit can always be defined since $\top : \Omega$ is required to be a Σ-proposition and 2) monad multiplication can always be defined because Σ-propositions are closed under dependent sums **(7.2.1∗5)**. We write $\mathbb{L} = (\mathsf{L}, \eta, \mu)$ for the resulting lift monad.

(7.2.1∗15) The lift monad is also called the Σ*-partial map classifier* because every Σ-partial map $A \hookleftarrow U \to B$ appears as the pullback of a unique map $A \to \mathsf{L}B$ as follows.

$$\begin{tikzcd}
U \arrow[r] \arrow[d, hook] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, "\eta_B"] \\
A \arrow[r] & \mathsf{L}B
\end{tikzcd}$$

(7.2.1∗16) Given a partial element $e : \mathsf{L}A$, we write $e{\downarrow} : \Sigma$ for its support, *i.e.* $-{\downarrow}$ is the first projection $\mathsf{L}(A) \to \Sigma$. When it is known that $e{\downarrow}$ holds, we may write $e : A$ for the defined element.

7.2.2. Phoa's principle.

(7.2.2∗1) So far we have not constrained the possible choices for Σ aside from properties needed to ensure that Σ-partial maps are closed under identity and composition. However, neither of the two obvious, extreme choices for the dominance necessarily leads to suitable models of axiomatic domain theory.

On the one hand, we may consider $\Sigma = 2$, which means we consider only decidable partial maps whose termination support is a decidable proposition. In this case the induced lift functor

$\mathsf{L}(A) = \Sigma_{\phi:2}.\phi \to A$ is equivalent to the functor $1+-$, and thus the initial lift algebra is the natural numbers object $\mathbb{N}$ and the final lift coalgebra is the extended natural numbers $\mathbb{N}_\infty$. Consequently the limit-colimit coincidence will likely fail because we do not have $\mathbb{N} \cong \mathbb{N}_\infty$ in most standard categories of predomains.

On the other hand, in a topos we may consider the class of *all* partial maps by taking $\Sigma = \Omega$ to be the subobject classifier. Unlikely the situation in **(7.2.1∗12)**, this choice will also fail because the induced Σ-partial map classifier admits maps that are *not* continuous — namely we may define a function $\neg : \Sigma \to \Sigma$ that takes the complement of the termination support of its input.

In this section we introduce an axiom called *Phoa's principle* that eliminates such pathological choices for the dominance and also unifies the two domain-theoretic order-like relations discussed in **(7.2.1∗3)**.

(7.2.2∗2) In the following we fix a Cartesian closed category $\mathscr{C}$ equipped with a dominance $(\Sigma, \top)$.

(7.2.2∗3) A *path* in an object A is a map $\Sigma \to A$. The *boundary* of a path $f : \Sigma \to A$ is a pair $\partial f : 1 \to A \times A$ given by evaluating at the distinguished points $\bot, \top : 1 \to \Sigma$.

(7.2.2∗4) It may be natural to wonder why the dominance Σ, which classifies the termination support of partial maps, is also used to classify paths. The reason is that in the models of axiomatic domain theory we consider the dominance indeed classifies paths. For instance, in the category of dcpos internal to a presheaf topos **(6.2.2∗1)**, we take the dominance to be $\Sigma = \Omega$, where Ω is the subobject classifier of the ambient topos. Recalling from **(6.3∗7)** that Ω is the free dcpo-completion of the interval poset $\mathbb{I}$, we see that continuous maps $\overline{f} : \Omega \to A$ are in bijective correspondence with monotone maps $f : \mathbb{I} \to A$, which is of course just a path $f(0) \sqsubseteq f(1)$ in A.

(7.2.2∗5) Let the subobject E be defined as the following equalizer:

$$\begin{tikzcd} E \arrow[r, hook] & \Sigma \times \Sigma \arrow[r, bend left, "\pi_1"] \arrow[r, bend right, "\wedge"'] & \Sigma \end{tikzcd}$$

We say $\mathscr{C}$ satisfies *Phoa's principle* when the boundary map $\partial : \Sigma^\Sigma \to \Sigma \times \Sigma$ factors through E via an isomorphism followed by an inclusion: $\Sigma^\Sigma \cong E \hookrightarrow \Sigma \times \Sigma$.

(7.2.2∗6) In a topos, the subobject E can be expressed as the subset $\{(\phi, \psi) \mid \phi \sqsubseteq \psi\}$ of pairs of Σ-propositions ordered by entailment. Thus Phoa's principle states that $f(\bot) \sqsubseteq f(\top)$ for any $f : \Sigma \to \Sigma$, and that every ordered pair $\phi \sqsubseteq \psi$ determines a map $\Sigma \to \Sigma$, with the two operations inverse to each other.

(7.2.2∗7) Thinking about the dominance in a category of domains as classifying observations in the sense of **(6.3∗4)**, Phoa's principle implies that one may *not* negate observations. In the context of recursion theory, this corresponds to the fact that the class of semi-decidable propositions (which form the Rosolini dominance **(7.2.1∗13)**) is not closed under complement. We will also use Phoa's principle in Section 7.3.4 and **(7.3.6∗6)** to develop a synthetic domain theory in which the path relation and specialization preorder on predomains coincide.

(7.2.2∗8) In the category of posets the interval dominance $\mathbb{I}$ satisfies Phoa's principle: the boundary of the three monotone functions $\mathbb{I} \to \mathbb{I}$ corresponds to ordered pairs $(0,0)$, $(0,1)$, and $(1,1)$.

(7.2.2∗9) In the category of internal dcpos, the dominance of **(7.2.1∗12)** given by the subobject classifier Ω satisfies Phoa's principle. This is because Ω is the free dcpo-completion or *ind-completion* of the interval $\{0 \sqsubseteq 1\}$. More precisely, de Jong [31, Example 4.9.1] shows that the family $\{\bot, \top\}$ is a (small) compact basis for Ω, thence we have that a continuous map $\Omega \to \Omega$ corresponds to a unique monotone map $\{\bot, \top\} \to \Omega$, *i.e.* a pair (ϕ, ψ) with $\phi \to \psi$.

7.2.3. Inductive fixed-point object.

(7.2.3∗1) In this section we outline the main infinitary axiom of axiomatic domain theory: the existence of a *inductive fixed-point object*, an object evincing a limit-colimit coincidence that allows one to define a fixed-point operator in any complying category.

(7.2.3∗2) Given a functor $F : \mathscr{C} \to \mathscr{C}$, an *$F$-invariant* object is a fixed-point of F, *i.e.* an object X such that $X \cong F(X)$.

(7.2.3∗3) An F-invariant object is *free* when it is both an initial F-algebra and final F-coalgebra.

(7.2.3∗4) Suppose that $\mathscr{C}$ possesses an initial lift algebra ω and a final lift coalgebra $\overline{\omega}$. Thinking of the canonical inclusion $\omega \hookrightarrow \overline{\omega}$ as the figure shape in which the generic chain ω is incident with the generic chain equipped with a top element $\overline{\omega}$, the main infinitary axiom of axiomatic domain theory is concentrated in the property that from the perspective of a category of predomains one has that the incidence relation/inclusion $\omega \hookrightarrow \overline{\omega}$ is an isomorphism, *i.e.* that $\overline{\omega}$ is a free L-invariant object.

(7.2.3∗5) *NB* — in the current context a lift algebra refers to an algebra for the *functor* L as opposed to the associated lift monad $\mathbb{L}$. Recall from **(3.2.4∗2)** that the latter notion comes with coherence laws that are not required of algebras of endofunctors.

(7.2.3∗6) Note that every lift algebra $\alpha : \mathsf{L}A \to A$ gives rise to a *successor* map $\sigma_\alpha : A \xrightarrow{\eta} \mathsf{L}A \xrightarrow{\alpha} A$. We write σ and $\overline{\sigma}$ for the successor maps associated to the initial lift algebra and final lift coalgebra, respectively.

(7.2.3∗7) Recall from **(6.1.1∗2)** that the lift $D_\bot$ of an ω-cpo D adjoins a distinguished element $\bot$ below all existing elements of D. The initial algebra ω for the lift functor $\bot$ is the canonical infinite chain equipped with a point at infinity: $\{0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq \infty\}$. The algebra map $\omega_\bot \to \omega$ is defined as follows.

$$\begin{aligned}
&\{\bot \sqsubseteq 0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq \infty\} \to \{0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq \infty\}\\
&\bot \mapsto 0\\
&n \mapsto n+1\\
&\infty \mapsto \infty
\end{aligned}$$

Given a lift algebra $\alpha : A_\bot \to A$, we may define an algebra morphism $\omega \to A$ as follows.

$$n \mapsto \sigma_\alpha^{(n)}(\alpha(\bot))$$

$\infty \mapsto \bigvee_{i=0} \sigma_\alpha^{(n)}(\alpha(\bot))$

In the above we write $\eta : A \to A_\bot$ for the inclusion of A into $A_\bot$. Observe that the finite prefixes of this map are ω-directed because we always have $\bot \sqsubseteq \eta(a)$ for any $a : A$.

(7.2.3∗8) In ω**CPO** ω is also the final lift coalgebra $\overline{\omega}$, with the coalgebra map $\overline{\omega} \to \overline{\omega}_\bot$ defined as follows.

$0 \mapsto \bot$
$n + 1 \mapsto n$
$\infty \mapsto \infty$

Given a lift coalgebra $\alpha : A \to A_\bot$, the universal map $A \to \overline{\omega}$ is defined by sending a to the number of times one can apply α before encountering $\bot$; if this number does not exist a is sent to ∞.

(7.2.3∗9) The point $\infty : 1 \to \overline{\omega}$ determined by universal coalgebra map out of the L-coalgebra $\eta_1 : 1 \to \mathsf{L}1$ is a fixed-point of $\overline{\sigma}$ [99].

(7.2.3∗10) In ω**CPO**, the successor map on $\omega \cong \overline{\omega}$ is the ordinary successor map extended by sending ∞ to itself. The point $\infty : 1 \to \overline{\omega}$ determined in **(7.2.3∗9)** is the actual point at infinity ∞ and is a σ-invariant point.

(7.2.3∗11) In $\mathscr{C}$ the *standard chain* is defined by iterating the lift functor:

$$\begin{tikzcd} \emptyset \arrow[r, "!"] & \mathsf{L}\emptyset \arrow[r, "L!"] & \mathsf{L}^2\emptyset \arrow[r, "L^2!"] & \ldots \end{tikzcd}$$

(7.2.3∗12) In ω**CPO** the initial lift algebra ω/final lift coalgebra $\overline{\omega}$ can be computed as the colimit of the standard chain, which is just inclusion of the finite prefixes of ω:

$$\begin{tikzcd} \emptyset \arrow[r, hook] & \{0\} \arrow[r, hook] & \{0 \sqsubseteq 1\} \arrow[r, hook] & \ldots \end{tikzcd}$$

(7.2.3∗13) An *inductive fixed-point object* [38] in $\mathscr{C}$ is an L-invariant object X that may be computed as the colimit of the standard chain. In Fiore and Plotkin [38] this inductive characterization of X is the primary ingredient in connecting the domain-theoretic structure on a base category with the synthetic domain theory of a topos.

(7.2.3∗14) Since the structure map of every initial algebra/final coalgebra is an isomorphism, by **(7.2.3∗12)** we have that the initial lift algebra/final lift coalgebra in ω**CPO** is an inductive fixed-point object.

7.2.4. Basic domain-theoretic category.

(7.2.4∗1) A *basic domain-theoretic category* (BDTC) is a Cartesian closed category $\mathscr{C}$ equipped with the following structure:

1. A dominance Σ satisfying Phoa's principle.

2. A *free* inductive fixed-point object $\infty : 1 \to \overline{\omega}$, *i.e.* an inductive fixed-point object that is both an *initial* L-algebra and a *final* L-coalgebra.

An object in a BDTC may be called a predomain, and an $\mathbb{L}$-algebra may be called a domain.

(7.2.4∗2) Every BDTC supports fixed-points of endomaps on domains; see for instance Reus and Streicher [99]. Briefly, for any $\mathbb{L}$-algebra $\alpha : \mathsf{L}A \to A$ and $f : A \to A$, we have a universal L-algebra morphism as follows.

$$\begin{tikzcd}
\mathsf{L}\overline{\omega} \arrow[r] \arrow[dd] & \mathsf{L}A \arrow[d, "\mathsf{L}f"] \\
 & \mathsf{L}A \arrow[d, "\alpha"] \\
\overline{\omega} \arrow[r, "\mathsf{kl}_f"] & A
\end{tikzcd}$$

The map $\mathsf{kl}_f : \overline{\omega} \to A$ can be thought of as an abstract version of the Kleene chain from **(6.1∗6)**; in fact when there is a nno in $\mathscr{C}$ the canonical inclusion $\mathbb{N} \to \overline{\omega}$ induces an ordinary Kleene chain $k : \mathbb{N} \to A$ with $k_n = f^{(n)}\bot_A$ [99].

(7.2.4∗3) To define the fixed-point of f, we observe the following commuting diagram.

$$\begin{tikzcd}
\overline{\omega} \arrow[r, "\mathsf{kl}_f"] \arrow[d, "\sigma"'] & A \arrow[d, "f"] \\
\overline{\omega} \arrow[r, "\mathsf{kl}_f"'] & A
\end{tikzcd} \qquad (7.2.4{*}3{*}1)$$

We then define the fixed-point of f to be the evaluation of the abstract Kleene chain at the invariant point $\infty : 1 \to \overline{\omega}$; the fixed-point property follows immediately by Eq. (7.2.4∗3∗1).

(7.2.4∗4) Fiore and Plotkin [38] base their notion of axiomatic domain theory around a *monadic base* $\mathscr{C}$ serving as a source of domain-theoretic enrichment. The Kleisli category $\mathscr{C}_{\mathbb{L}}$ is called a *Kleisli model of ADT* in *op. cit.* when it is $\mathscr{C}$-algebraically compact; this structure is primarily used to interpret the programming language **FPC**, an extension of **PCF** with recursive types. Because the denotational semantics of recursive types is outside the scope of this dissertation, we axiomatize a weaker notion of ADT that is essentially the monadic base of *op. cit.* without the symmetric monoidal closed structure on the category of $\mathbb{L}$-algebras $\mathscr{C}^{\mathbb{L}}$.

7.3. SYNTHETIC DOMAINS

(7.3∗1) In contrast to the relatively simple synthetic preorder theory of **(7.1.1∗4)**, the axiomatics of synthetic domain theory is much more involved. Conceptually we may split the definitions and axiomatic structure of this section into two parts: the domain-theoretic and the order-theoretic.

$$\begin{tikzcd}
\text{replete} \\ \textbf{(7.3.1*6)} \arrow[d] \arrow[dr] \arrow[drr, bend left] \arrow[ddrr, bend left] & & \text{Phoa's principle} \\ \textbf{(7.2.2*5)} \arrow[d] \\
\text{(well) complete} \\ \textbf{(7.3.1*3)} \arrow[d, "\text{Section 7.3.6.3}"] & \text{boundary separated} \\ \textbf{(7.3.3*6)} \arrow[d, "\textbf{(7.3.6.1*3)}"] & \text{linked} \\ \textbf{(7.3.4*1)} \arrow[dl] \arrow[d] \\
\text{closed under} \\ \text{synthetic } \omega\text{-joins} & \text{(path = intrinsic)} \\ \text{pointwise} & \text{(path = intrinsic)} \\ \text{partially ordered}
\end{tikzcd}$$

Figure 7.1: Relationships between various properties on types. Arrows represent implications; multiple premises to a node should be understood conjunctively. Unlabeled implications all refer to **(7.3.6*6)**. Note that in the middle right and lower right nodes the path relation and intrinsic preorder coincide from the premise of linkedness.

One the domain-theoretic side, we isolate a class of predomain-like types that "believe" in the limit-colimit coincidence. Technically this is achieved by requiring every predomain to be orthogonal **(3.5*2)** to the inclusion $\omega \hookrightarrow \overline{\omega}$.

On the order-theoretic side, we show that for a certain subset of predomain-like types to be defined in Section 7.3.1, the specialization/intrinsic preorder and path relation coincide; we call the resulting order relation the *synthetic order* on predomains, which furnishes a convenient *interface* for developing denotational semantics in a manner that resembles classic domain-theoretic semantics. Although all of the technical results of this thesis may be carried out without reference to the synthetic order on predomains, as argued in Niu, Sterling, and Harper [89], the synthetic order provides an intuitively appealing framework for doing classic denotational semantics without the drawback of constantly having to verify continuity side conditions. Moreover, synthetic (pre)orders have also been used by Grodin et al. in the context of *cost analysis*, which is germane to the broader picture of this dissertation. In this context an order-theoretic language appears to be the right abstraction for combining partiality and cost structure [66, Section 8].

(7.3*2) The plan is as follows. In Section 7.3.1 we define the class of *well-complete* types which can be thought of as the synthetic counterparts to ω-cpos. Although well-complete types suffice to develop the denotational semantics of general recursion, their order structure is not necessarily well-behaved: the path relation and intrinsic preorder need not coincide, and neither needs to be computed pointwise on limits or partially ordered. To ensure a well-behaved order structure on predomains we will work with a more restrictive class called the *replete* types. The relationships between the various classes of types are depicted in **(7.3*2)**.

(7.3*3) In this section we fix a topos $\mathscr{E}$ equipped with a dominance $(\Sigma, \top)$ satisfying Phoa's principle. We write $\mathbb{L} = (\mathsf{L}, \eta, \mu)$ for the induced lifting structure.

i **(7.3*4)** In this section by an orthogonal object or local object we mean orthogonality in the internal sense **(3.5∗3)**.

7.3.1. Complete types, replete types, synthetic predomains.

(7.3.1∗1) We synthesize the discussion of Section 7.2 on the axiomatic properties shared by concrete categories of domains and use them as criteria to isolate a class of types in $\mathscr{E}$ to serve as *synthetic* predomains. We first introduce the notion of *complete* types [39, 38], which can be thought of as the synthetic counterparts to the ω-cpos of classic domain theory.

(7.3.1∗2) A *synthetic ω-chain* in a type A is a map $\omega \to A$ out of the initial L-algebra.

(7.3.1∗3) A type A is *complete* when it is orthogonal to the canonical inclusion $\omega \to \overline{\omega}$, *i.e.* the following holds in an internal sense:

$$\begin{tikzcd}
\omega \arrow[r, tail] \arrow[d] & \overline{\omega} \arrow[dl, dotted, "\exists!"] \\
A &
\end{tikzcd}$$

A type A is *well complete* when $\mathsf{L}A$ is complete. In particular we have that every synthetic ω-chain in a complete type is equipped with a supremum given by evaluating at the point $\infty : 1 \to \overline{\omega}$.

(7.3.1∗4) To get a class of predomains that is workable for basic denotational semantics, Longley and Simpson introduced the notion of well-complete types as the least restrictive possible notion of predomain that is closed under lifting. In contrast we will consider the dual *most restrictive* class of predomain, the *replete* types [62], which has better properties with respect to the *synthetic order* of synthetic predomains.

(7.3.1∗5) A map $f : X \to Y$ is called Σ-*equable* or a Σ-*isomorphism* when Σ is orthogonal to f.

(7.3.1∗6) A type is *replete* when it is orthogonal to every Σ-isomorphism. A *predomain* is a replete type.

(7.3.1∗7) The intuition behind replete types is that they "believe" in every isomorphism from the perspective of Σ. Thinking about a map $A \to \Sigma$ as a computational predicate, a Σ-isomorphism $A \to B$ is an isomorphism "up to" computational properties. Moreover, observe that by definition every property of Σ that is defined by a localization with respect to a collection of maps is also shared by a replete type. We will use this fact to show that the intrinsic order on replete types is extremely well-behaved — in particular that it is partially ordered and pointwise on limits.

(7.3.1∗8) Both well-complete types and replete types are (internally) complete and cocomplete and form reflective exponential ideals in $\mathscr{E}$. Thus under either notion of predomains we have a standard interpretation of **PCF** by means of the given Cartesian closed structure (in Section 7.3.6 we explain how to interpret the fixed-point operator by means of the *synthetic ω-join* on predomains). The difference is that class of replete types may be universally characterized as the smallest reflective exponential ideal containing Σ, and thus we have that every replete object is also well-complete but not vice versa [80].

7.3.2. The intrinsic order.

(7.3.2∗1) For every type A, one may define the *intrinsic preorder* $\sqsubseteq^{\circ}_{A}$ on A analogous to the specialization preorder on a topological space **(6.3∗8)**: $x \sqsubseteq^{\circ}_{A} y$ if and only if $f\ x$ implies $f\ y$ for every $f : A \to \Sigma$.

(7.3.2∗2) Under the view of Σ-predicates $f : A \to \Sigma$ as observations **(6.3∗4)**, we may understand $x \sqsubseteq^{\circ}_{A} y$ as y is apprehended by every observation apprehending x, *i.e.* y contains at least as much computational information as x.

□ **(7.3.2∗3)** We have that $\bot \sqsubseteq^{\circ} \top$ on the intrinsic preorder.

■ **(7.3.2∗4)** Indeed, fixing a map $f : \Sigma \to \Sigma$, by Phoa's principle, evaluation at boundary obtains a pair $(f\ \bot, f\ \top)$ such that $f\ \bot$ implies $f\ \top$.

7.3.3. The path relation.

(7.3.3∗1) Unfortunately, the intrinsic order need not be well-behaved generally. For instance, $\sqsubseteq^{\circ}_{A}$ may fail to be a partial order, and the intrinsic order on limits need not be pointwise. To rectify this, we introduce the *path relation.*

(7.3.3∗2) The *path relation* $x \sqsubseteq^{\mathsf{p}}_{A} y$ is defined hold if and only if there exists a path **(7.2.2∗3)** $\Sigma \to A$ whose boundary is (x, y).

(7.3.3∗3) The path relation is an alternative way to surface the order structure of predomains, studied in much more detail by Fiore [36]; the path relation is also used by Grodin *et al.* [46] to obtain a theory of synthetic preorders for cost analysis.

□ **(7.3.3∗4)** The path relation need not coincide with the intrinsic order, but a path $\alpha : x \sqsubseteq^{\mathsf{p}} y$ always implies $x \sqsubseteq^{\circ} y$.

■ **(7.3.3∗5)** Suppose we have a path $l : \Sigma \to \Sigma$ whose boundary is determined by x, y. Fixing $f : \Sigma \to \Sigma$, we want to show that $f\ x \to f\ y$. But this follows by evaluating $\bot \sqsubseteq^{\circ} \top$ at the Σ-predicate $f \circ l : \Sigma \to \Sigma$.

(7.3.3∗6) A type is *boundary separated* when two paths in the type with identical boundaries are equal.

(7.3.3∗7) Consider the following pushout:

$$\begin{tikzcd}
2 \arrow[r, "{(\bot,\top)}"] \arrow[d, "{(\bot,\top)}"'] & \Sigma \arrow[d] \\
\Sigma \arrow[r] & S \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

By the universal property of S, we have that maps $S \to A$ correspond to paths in A with equal boundaries. There is a canonical map $s : S \to \Sigma$ given by the identity $1 : \Sigma \to \Sigma$ satisfying $\left(\Sigma \to S \xrightarrow{s} \Sigma\right) = \left(\Sigma \xrightarrow{1} \Sigma\right)$.

□ **(7.3.3∗8)** A type A is boundary separated when it is orthogonal to $s : S \to \Sigma$.

7.3.4. Linked types.

(7.3.4∗1) A type A is *linked* when the path relation on A coincides with the intrinsic order on A.

(7.3.4∗2) A number of good properties follow from the fact that a type is linked. In particular, we may use it to show that the intrinsic order on Σ = path relation on Σ = entailment order on Σ.

□ **(7.3.4∗3)** The dominance Σ is linked.

■ **(7.3.4∗4)** One direction already holds by **(7.3.3∗4)**. Conversely, if $x \sqsubseteq^\circ y$, then we have in particular $x \to y$, which by Phoa's principle uniquely determines a path $l : \Sigma \to \Sigma$.

(7.3.4∗5) The intrinsic order on Σ coincides with the entailment order.

(7.3.4∗6) In realizability models of synthetic domain theory, a linked object is always boundary separated, but this need *not* be the case in the synthetic domain theory we consider in this dissertation. The reason is that in the realizability setting one typically assumes *Markov's principle*, which has the effect of making the dominance Σ a $\neg\neg$-separated object. Therefore it is tenable to work, for instance in the effective topos, in the full subcategory of $\neg\neg$-separated predomains (*i.e.* assemblies), which are automatically boundary separated [94, Lemma 5.4.3]. Because we strive to be *constructive* rather than anti-classical, we do not rely on Markov's principle and thus need to separately stipulate boundary separation on linked objects.

(7.3.4∗7) Nonetheless, the dominance Σ itself is boundary separated by virtue of Phoa's principle.

7.3.5. A cost-sensitive model of synthetic domain theory.

(7.3.5∗1) To obtain an account of cost structure as in the manner Chapter 4, we will layer the FC phase distinction on top of the synthetic domain theory topos. From an internal perspective, this corresponds to assuming a distinguished Σ-proposition ¶. We require the phase distinction proposition to be computational in order to ensure that the associated sealing monad $\bullet-$ preserves the property of being a predomain. In Chapter 8 we use this property to define a denotational model of a *cost-sensitive* version of **PCF**.

(7.3.5∗2) An *SDT model of the FC phase distinction* consists of a topos $\mathscr{E}$ equipped with a *complete* dominance Σ satisfying Phoa's principle and a Σ-proposition ¶. Moreover, we require that the type of booleans 2 is a predomain and restriction-modal in the sense of **(3.3∗7)**.

7.3.6. Properties of predomains.

(7.3.6∗1) In the following we assume an SDT model of the FC phase distinction $(\mathscr{E}, \Sigma, ¶)$.

(7.3.6∗2) We wish to establish the following properties of the intrinsic preorder on predomains:

1. The intrinsic preorder is pointwise on products, functions, and liftings of predomains.
2. The intrinsic preorder on a predomain is a *synthetic ω-complete partial order*.

3. The synthetic ω-complete partial order structure of predomains is defined componentwise for products and functions between predomains.

(7.3.6∗3) The general properties of the intrinsic preorder and path relation in SDT have been investigated in several prior works [94, 98, 77]. In this dissertation we merely organize and collate the known results under a single framework.

(7.3.6∗4) Consider the following pushout (note that it is different from the one in **(7.3.3∗7)** since the left map is $(\top, \bot)$ instead of $(\bot, \top)$).

$$\begin{tikzcd}
2 \arrow[r, "{(\bot,\top)}"] \arrow[d, "{(\top,\bot)}"'] & \Sigma \arrow[d] \\
\Sigma \arrow[r] & D \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

The object D has the following universal property: a map $D \to A$ is given by a pair of paths $f, g : \Sigma \to A$ such that f and g have swapped boundaries, *i.e.* we have that $f : a \sqsubseteq b$ and $g : b \sqsubseteq a$.

□ **(7.3.6∗5)** The path relation on a type A is anti-symmetric when A is orthogonal to the terminal map $D \to 1$.

□ **(7.3.6∗6)** Any predomain A enjoys the following properties:

1. *Completeness*: A is orthogonal to $\omega \hookrightarrow \overline{\omega}$.

2. *Anti-symmetry*: the intrinsic preorder on A is a partial order.

3. *Boundary separation*: maps $\Sigma \to A$ with equal boundary are equal.

4. *Linkedness*: the intrinsic preorder and the path relation on A coincide.

Because of linkedness, we may speak of a single *synthetic order* $\sqsubseteq$ on any predomain A.

■ **(7.3.6∗7)** By **(7.3.1∗7)** replete types are both complete and boundary separated they are defined by localizations (by **(7.3.1∗3)** and **(7.3.3∗8)**). Assuming Phoa's principle, the proof that replete types are linked can be found in Taylor [127, Corollary 2.10] and Reus [98, Corollary 6.1.16]. Lastly, by **(7.3.6∗5)**, objects whose path relation is antisymmetric can be defined as a localization, thus we know that the link relation on every replete type is antisymmetric, hence it follows the intrinsic order on replete types are also partial orders since they are linked.

7.3.6.1. The synthetic order on predomains.

□ **(7.3.6.1∗1)** Every map $f : A \to B$ between predomains is monotone with respect to the synthetic order.

■ **(7.3.6.1∗2)** Given $a \sqsubseteq a'$, we derive a path $\Sigma \to B$ whose boundary is $(f\ a, f\ a')$ by postcomposing with f, and so $f\ a \sqsubseteq f\ a'$ as well.

□ **(7.3.6.1∗3)** The synthetic orders on products and exponentials of predomains are pointwise.

■ **(7.3.6.1∗4)** This is proven by Phoa [94, Proposition 5.4.4]. We recall the case for the function types. Given a path $f \sqsubseteq_{X\to B} g$, it is clear that we may construct a path $f\ x \sqsubseteq_B g\ x$ for all $x : X$. Conversely, suppose we are given a path $f\ x \sqsubseteq_B g\ x$ for all $x : X$. By **(7.3.6∗6)**, B is boundary separated, and so such paths are necessarily unique, and so we have a function $\alpha : X \to \Sigma \to B$ such that $\alpha(x)$ is a path $f\ x \sqsubseteq Bg\ x$. We then obtain a path $f \sqsubseteq_{X\to B} g$ by taking the exponential transpose of α.

□ **(7.3.6.1∗5)** Given a predomain A, we have that $x \sqsubseteq_{\mathsf{L}A} y$ if and only if $x{\downarrow}$ implies $y{\downarrow}$ and whenever $x{\downarrow}$, we have $x \sqsubseteq_A y$.

■ **(7.3.6.1∗6)** In the forward direction, we have a path $x{\downarrow} \sqsubseteq y{\downarrow}$, which means $x{\downarrow}$ implies $y{\downarrow}$ as the Σ is linked by **(7.3.4∗3)**. Suppose $x{\downarrow}$ holds, and let $f : A \to \Sigma$ be arbitrary. By assumption, we have that $f'(x) \to f'(y)$, where $f'((\phi, u)) = \phi \angle f \circ u$. In other words, we have $x{\downarrow} \angle f(x)$ implies $y{\downarrow} \angle f(y)$. Since $x{\downarrow}$ holds, we have that $f(x) \to f(y)$, which by definition means $x \sqsubseteq_A y$.

In the backward direction, let $\alpha : x{\downarrow} \to x \sqsubseteq_A y$ be the given partial path. We may define a total path $\beta : \Sigma \to \mathsf{L}A$ between x and y by setting $\beta(\phi) = (x{\downarrow}, \lambda p.\ \alpha\ p\ \phi)$. Thus we have $x \sqsubseteq_{\mathsf{L}A} y$ as required.

7.3.6.2. Discrete predomains and Σ-equality.

🕮 **(7.3.6.2∗1)** A type A is called *flat* or *discrete* when $x \sqsubseteq^{\circ} y$ implies $x = y$.

🕮 **(7.3.6.2∗2)** A type *has* Σ*-equality* when its equality relation is valued in Σ-propositions.

□ **(7.3.6.2∗3)** Any type with Σ-equality is discrete.

■ **(7.3.6.2∗4)** Let $f : A \to \Sigma$ be the characteristic map that sends a to $a = x$, which by assumption is a Σ-proposition. Since $x \sqsubseteq_A y$ on the specialization order and $f(x)$ holds, we have that $f(y)$ holds as well.

⚠ **(7.3.6.2∗5)** The category of predomains possesses a natural numbers type $\mathbb{N}_{\mathsf{P}}$ with Σ-equality, which means it is also discrete. Note that it is not necessarily the same as the ambient natural numbers type $\mathbb{N}$, and we will not assume that it is the case in this chapter or Chapter 8. From a logical perspective, this difference means that $\mathbb{N}_{\mathsf{P}}$ has a universal mapping-out property whose motive is valued in predomains rather than arbitrary types. Nonetheless the natural numbers object of predomains plays an important role in other approaches to computational adequacy in synthetic domain theory, which we discuss in Section 8.5.

□ **(7.3.6.2∗6)** If A has Σ-equality, then so does $¶ \vee A$.

■ **(7.3.6.2∗7)** We observe that $¶ \vee A$ is defined as the pushout of the projections of $A \times ¶$ as indicated below:

$$\begin{tikzcd}
A \times ¶ \arrow[r] \arrow[d] & A \arrow[d, "\eta_{¶\vee -}"] \\
¶ \arrow[r, "\star"'] & ¶ \vee A \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

Using the fact that A has Σ-equality, we obtain a map $f : (\P \vee A) \times (\P \vee A) \to \P \vee \Sigma$ such that $f(\eta_{\P\vee -}(x), \eta_{\P\vee -}(y)) = \eta_{\P\vee -}(x = y)$ and $f(\star, -) = f(-, \star) = \star$. The desired characteristic map can then be defined as $\sigma \circ f$, where $\sigma : \P \vee \Sigma \to \Sigma$ is defined as follows.

$$\begin{aligned}
&\sigma : \P \vee \Sigma \to \Sigma \\
&\sigma(\eta_{\P\vee -}(\phi)) = \P \vee \phi \\
&\sigma(\star) = \top
\end{aligned}$$

For any $u, v : \P \vee A$, if $u = v = \eta_{\P\vee -}(x)$ for some $x : A$, then we have $\sigma(f(u, v)) = \sigma(\eta_{\P\vee -}(x = x)) = \P \vee (x = x) = \top$. Otherwise, we have that u or v is $\star$, which means $\P$ holds and so $\sigma(f(u, v)) = \sigma(\star) = \top$ as well. Conversely, suppose $\sigma(f(u, v)) = \top$, and that $u = \eta_{\P\vee -}(x)$ and $v = \eta_{\P\vee -}(y)$, which means that $\sigma(\eta_{\P\vee -}(x = y)) = \P \vee (x = y)$ holds. If $\P$ holds, we are done as $(\P \vee A) \cong 1$ in this case. Otherwise, we have $x = y$, and so $u = \eta_{\P\vee -}(x) = \eta_{\P\vee -}(y) = v$. Lastly, if either u or v is the unique element $\star$ then we may discharge the case as above.

7.3.6.3. The synthetic ω-complete partial order structure.

☐ **(7.3.6.3∗1)** In a topos with a dominance Σ and a natural numbers object $\mathbb{N}$, the initial lift algebra ω and final lift coalgebra $\overline{\omega}$ can be characterized as subsets of $\mathbb{N} \to \Sigma$ [130]. Explicitly we have the following formula for the underling sets of $\overline{\omega}$:

$$\left\{F : \Sigma^{\mathbb{N}} \mid \forall[n : \mathbb{N}] \;\; F(n+1) \to F(n)\right\}$$

The invariant point $\infty : 1 \to \overline{\omega}$ is defined as the constant map $\mathbb{N} \to \Sigma$ determined by $\top : \Sigma$.

☐ **(7.3.6.3∗2)** The invariant point is the top element of $\overline{\omega}$ with respect to the intrinsic preorder.

■ **(7.3.6.3∗3)** Observe that for any $i : \overline{\omega}$ we have a path $\alpha : i \sqsubseteq^{\mathsf{p}} \infty$ defined by $\alpha(\phi, n) = \phi \vee i(n)$. Thus we have $i \sqsubseteq^{\circ} \infty$ by **(7.3.3∗4)**.

☐ **(7.3.6.3∗4)** For any $\alpha : A$ the principal lower set $\downarrow\alpha$ of a complete type A is complete.

■ **(7.3.6.3∗5)** We can express the principal lower set as follows.

$$\begin{aligned}
\downarrow(\alpha) &= \{a \mid a \sqsubseteq^{\circ} \alpha\} \\
&= \{a \mid \forall f : A \to \Sigma \; f(a) \to f(\alpha)\} \\
&= \bigcap_{f : A \to \Sigma} \{a \mid f(a) \to f(\alpha)\}
\end{aligned}$$

Because complete types are internally complete, the result would follow if we can show that $S = \{a \mid f(a) \to f(\alpha)\}$ is complete. We may show that S can be computed as follows.

$$\begin{tikzcd}
S \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "{\langle f, f(\alpha)\rangle}"] \\
\Sigma^{\Sigma} \arrow[r, "\partial"'] & \Sigma \times \Sigma
\end{tikzcd}$$

Since S can be defined as the limit of a diagram of complete types, it is complete as well.

☐ **(7.3.6.3∗6)** For every map $f : \omega \to A$ into a complete type A, there exists an element $f_\infty : A$ such that f_∞ is a least upper bound of f with respect to the intrinsic preorder.

■ **(7.3.6.3∗7)** Define f_∞ be the element determined by the unique extension $\overline{f} : \overline{\omega} \to A$ evaluated at the invariant point $\infty : 1 \to \overline{\omega}$.

1. First we show that f_∞ is an upper bound for f. Fixing $i : \omega$, we need to show that $f\ i \sqsubseteq^\circ_A f_\infty$. Because $\overline{f}$ extends f, it suffices to show $\overline{f}\ i \sqsubseteq^\circ_A f_\infty$. Using the fact that every map is monotone with respect to the intrinsic preorder, the result follows from **(7.3.6.3∗2)**.

2. Let α be an upper bound for f. We need to show that $f_\infty \sqsubseteq^\circ \alpha$. By **(7.3.6.3∗4)** the principal lower set $\downarrow(\alpha)$ is complete, so we have the following lifting property:

$$\begin{tikzcd}
\omega \arrow[rr, hook] \arrow[dr, "f"'] & & \overline{\omega} \arrow[dl, dashed, "\tilde{f}"] \\
& {\downarrow(\alpha) = \{a \mid a \sqsubseteq^\circ \alpha\}} &
\end{tikzcd}$$

In the above $\tilde{f}$ is the unique extension of f considered as a map $\omega \to \downarrow(\alpha)$. By uniqueness of $\overline{f}$ as the extension of $f : \omega \to A$, $\tilde{f}$ is equal to $\overline{f}$ considered as maps $\overline{\omega} \to A$. Consequently we have that $f_\infty = \overline{f}(\infty) = \tilde{f}(\infty)$, so the result follows by observing that $\tilde{f}(\infty) \in \downarrow(\alpha)$.

︾ **(7.3.6.3∗8)** Because the intrinsic preorder on every predomain A is a partial order, every synthetic ω-chain $f : \omega \to A$ is equipped with a *synthetic ω-join*, which we write as $\bigvee f : A$.

☐ **(7.3.6.3∗9)** Every map $f : A \to B$ between predomains is continuous, in the sense that f preserves synthetic ω-joins.

■ **(7.3.6.3∗10)** Fix a synthetic ω-chain $d : \omega \to A$. We need to show that $f(\bigvee d) = \bigvee(fd)$. By Section 7.3.6 predomains are complete, so we have the following extensions of d and $f \circ d$:

$$\begin{tikzcd}
\omega \arrow[r] \arrow[d, hook] & A \arrow[d, "f"] \\
\overline{\omega} \arrow[ur, dashed, "\overline{d}"] \arrow[r, dashed, "\overline{f \circ d}"'] & B
\end{tikzcd}$$

Because extensions along $\omega \hookrightarrow \overline{\omega}$ are unique for complete types, we have $f \circ \overline{d} = \overline{f \circ d}$. But by definition of the synthetic ω-join, this means that $f(\bigvee d) = f(\overline{d}(\infty)) = \overline{f \circ d}(\infty) = \bigvee(f \circ d)$.

7.3.7. Domains and recursion.

📖 **(7.3.7∗1)** A *domain* is a predomain equipped with a $\mathbb{L}$-algebra structure.

☐ **(7.3.7∗2)** A predomain is a domain if and only if it has a least element [36, 117].

(7.3.7∗3) Analogous to the discussion in Section 7.2.4 in the context of axiomatic domain theory, synthetic domains also support recursive functions by taking the synthetic ω-join of the abstract Kleene chain of a domain endomap. The main difference is that we need to take care to distinguish ω from $\overline{\omega}$ since they are different from the perspective of general types. In the following we recap the fixed-point construction of Reus and Streicher [99].

🔨 **(7.3.7∗4)** For every domain $\alpha : \mathsf{L}D \to D$ and $f : D \to D$, define the *Kleene chain* of f as the universal algebra morphism in the following situation:

$$\begin{tikzcd}
\mathsf{L}\omega \arrow[r] \arrow[d] & \mathsf{L}D \arrow[d, "\alpha Lf"] \\
\omega \arrow[r, dotted, "\mathsf{kl}_f"'] & D
\end{tikzcd} \qquad (7.3.7{*}4{*}1)$$

Because (pre)domains are complete **(7.3.6∗6)**, there is a unique extension $\overline{\mathsf{kl}_f} : \overline{\omega} \to D$:

$$\begin{tikzcd}
\omega \arrow[r, tail] \arrow[d] & \overline{\omega} \arrow[dl, dotted] \\
D &
\end{tikzcd}$$

□ **(7.3.7∗5)** Every map $f : D \to D$ with D a domain has a fixed-point.

■ **(7.3.7∗6)** We define the fixed-point of f to be $\bigvee \mathsf{kl}_f : D$, which by **(7.3.6.3∗7)** is defined to be $\overline{\mathsf{kl}_f}(\infty)$. Recalling from **(7.2.3∗9)** that $\infty : \overline{\omega}$ is invariant under the successor map $\overline{\sigma}$, it suffices to show that the following diagram commutes:

$$\begin{tikzcd}
\overline{\omega} \arrow[r] \arrow[d, "\overline{\sigma}"'] & D \arrow[d, "f"] \\
\overline{\omega} \arrow[r] & D
\end{tikzcd}$$

Since D is a complete object, it suffices to show that both maps $\overline{\omega} \to D$ arise as the corresponding extensions of $\omega \to D$, *i.e.* the following diagram commutes:

$$\begin{tikzcd}
\omega \arrow[r, tail] \arrow[d, "\sigma"'] \arrow[rr, bend left, "\mathsf{kl}_f"] & \overline{\omega} \arrow[r] \arrow[d, "\overline{\sigma}"'] & D \arrow[d, "f"] \\
\omega \arrow[r, tail] \arrow[rr, bend right, "\mathsf{kl}_f"'] & \overline{\omega} \arrow[r] & D
\end{tikzcd}$$

But this follows by Eq. (7.3.7∗4∗1) and recalling that $\sigma = \omega \xrightarrow{\eta_{\mathbb{L}}} \mathsf{L}\omega \to \omega$.

7.3.7.1. Admissibility.

📖 **(7.3.7.1∗1)** A subset of a domain is *admissible* when it is complete and closed under the least element.

☐ **(7.3.7.1∗2)** The intersection of a family of admissible subsets of a domain is admissible.

■ **(7.3.7.1∗3)** The least element is contained in the intersection as it is contained in every fibre. That complete types are closed under intersections essentially follows from Simpson [112, Proposition 2.5] using the fact that completeness can be expressed using the internal language.

☐ **(7.3.7.1∗4)** If P, Q are Σ-subsets of a domain A with $\bot \in P \to \bot \in Q$, then the exponential subobject Q^P is an admissible subset of A.

■ **(7.3.7.1∗5)** We have that $\bot \in Q^P$ by the premise, and by an argument similar to **(7.3.6.3∗5)**, we have that Q^P is complete as well. In particular, we observe that Q^P may be defined as the following pullback of complete types:

$$\begin{tikzcd}
Q^P \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "\langle P, Q\rangle"] \\
\Sigma^\Sigma \arrow[r, "\partial"'] & \Sigma \times \Sigma
\end{tikzcd}$$

☐ **(7.3.7.1∗6)** We may establish admissible properties of recursive functions by *fixed-point induction* [99, Theorem 8.2]: given an admissible subset $A \subseteq D$ and $f : D \to D$ such that f restrict to a map $A \to A$, we have that $\mathsf{fix}(f) \in A$.

☐ **(7.3.7.1∗7)** The fixed-point defined in **(7.3.7∗5)** is the least (pre)-fixed-point with respect to the intrinsic order [99, Theorem 8.18].

∗ ∗ ∗

⚠ **(7.3.7.1∗8)** In Niu, Sterling, and Harper [89] I had mistakenly defined an admissible subset to be any subset closed under $\bot$ and synthetic ω-joins. Although a natural and tempting generalization of classic admissibility, this definition neglects the fact that constructively one does not have a case analysis principle for $\overline{\omega}$ distinguishing ∞ from those in the image of $\omega \hookrightarrow \overline{\omega}$. This corresponds to the fact that $\omega \cup \{\infty\} \hookrightarrow \overline{\omega}$ is $\neg\neg$-dense, *i.e.* $\overline{\omega}$ consists of elements i such that it is false that i is neither ∞ nor contained in ω. Consequently, the admissibile subsets $A \hookrightarrow D$ proposed in *op. cit.* only satisfy fixed-point induction when the corresponding characteristic proposition $d \in A$ is $\neg\neg$-closed.

(7.3.7.1∗9) Nonetheless, all concrete instances of admissible subsets used in Niu, Sterling, and Harper [89] are admissible in the sense of **(7.3.7.1∗1)** as we show in **(7.3.7.1∗10)**.

☐ **(7.3.7.1∗10)** A *lower* subset of a complete type is complete when it is closed under synthetic ω-joins.

■ **(7.3.7.1∗11)** Suppose $S \subseteq A$ is downward closed and closed under synthetic ω-joins. We must complete the following lifting problem:

$$\begin{tikzcd}
\omega \arrow[r, tail] \arrow[d, "f"'] & \overline{\omega} \arrow[dl, dotted, "?"] \\
S &
\end{tikzcd}$$

Because S is complete, we have the following extension of $\omega \xrightarrow{f} A \xhookrightarrow{\iota} S$:

$$\begin{tikzcd}
\omega \arrow[r, tail] \arrow[d, "f"'] & \overline{\omega} \arrow[dl, dotted, "?"] \arrow[ddl, dotted, "\overline{\iota f}"] \\
S \arrow[d, tail] & \\
A &
\end{tikzcd}$$

Thus it suffices to show that $\overline{\iota f}$ factors through S. Fixing $i : \overline{\omega}$, we want to show that $\overline{\iota f}(i) \in S$. By the assumptions on S, we have that $\bigvee(\iota f) = \overline{\iota f}(\infty)$ is in S and that $a \in S$ whenever $a \sqsubseteq \overline{\iota f}(\infty)$. Thus it suffices to show $\overline{\iota f}(i) \sqsubseteq \overline{\iota f}(\infty)$, which follows from **(7.3.6.3∗2)**.

7.4. A RELATIVE SHEAF MODEL OF COST-SENSITIVE SDT

(7.4∗1) In this section we substantiate the internal developments of Section 7.3 by means of a model construction based on the *relative sheaf model of SDT* of Sterling and Harper [122]. In fact the results of *op. cit.* show that every basic domain-theoretic category in the sense of **(7.2.4∗1)** internal to a presheaf topos $\mathscr{E}$ lifts to a model of synthetic domain theory fibred over $\mathscr{E}$. Sterling and Harper [122] do not rely on some properties of the resulting SDT such as repleteness or Phoa's principle, but they nonetheless hold in the models so constructed. In the following we recall some key definitions and results from *op. cit.* and verify that every internal BDTC indeed lifts to a model of cost-sensitive synthetic domain theory in the sense of **(7.3.5∗2)**.

(7.4∗2) Because we want to model the FC phase distinction, we fix $\mathscr{E} = \widehat{\mathbb{I}}$ and consider a BDTC $\mathscr{D}$ internal to $\widehat{\mathbb{I}}$. Sterling and Harper [122] consider the category of internal dcpos of $\widehat{\mathbb{I}}$ as in **(6.2.2∗1)**, but a suitable site may be constructed out of any *extensive* **(3.4.2∗14)** BDTC.

(7.4∗3) Recall that a BDTC $(\mathscr{D}, \Sigma)$ consists of a dominance satisfying Phoa's principle and a free inductive fixed-point object $\omega \cong \overline{\omega}$. In the following, we assume that $\mathscr{D}$ also contains the intermediate proposition $\P : \Omega_{\widehat{\mathbb{I}}}$ determined by $\underset{1}{\overset{0}{\downarrow}}$.

☐ **(7.4∗4)** Any extensive BDTC as in **(7.4∗3)** embeds into a sheaf topos $\mathrm{Sh}(\mathscr{D})$ for the extensive coverage in which $\Sigma = \mathsf{y}_{\mathscr{D}}(\Sigma)$ is a complete dominance closed under finite joins and the proposition $¶ = \mathsf{y}_{\mathscr{D}}¶$ is a Σ-proposition [122].

☐ **(7.4∗5)** The dominance Σ in the sheaf topos $\mathrm{Sh}(\mathscr{D})$ satisfies Phoa's principle.

■ **(7.4∗6)** Recalling from **(7.2.2∗5)** that we seek to factor the boundary map $\partial : \Sigma^{\Sigma} \to \Sigma \times \Sigma$ through $\Sigma^{\Sigma} \cong E \hookrightarrow \Sigma \times \Sigma$, the result follows from the fact that $\mathsf{y} : \mathscr{D} \to \mathrm{Sh}(\mathscr{D})$ is a Cartesian closed functor and thus preserves the map ∂; moreover recalling that E is defined as the equalizer of $\Sigma \times \Sigma \xrightarrow{\pi_1, \wedge} \Sigma$, we also have $E = \mathsf{y}E$ in $\mathrm{Sh}(\mathscr{D})$ since y preserves limits as well.

☐ **(7.4∗7)** The constant presheaf 2 determined by $2 : \mathbf{Set}$ is $(¶ \to -)$-modal (*i.e.* restriction-modal).

■ **(7.4∗8)** We need to show that 2 is internally orthogonal to $¶ \to 1$, *i.e.* we have the following unique lifting property:

$$\begin{tikzcd}
X \times ¶ \arrow[r, tail] \arrow[d] & X \times 1 \arrow[dl, dotted, "?"] \\
2
\end{tikzcd}$$

Because a presheaf X is a canonically a colimit of representables **(3.1.4∗3)** and the functor $- \times Z$ preserves colimits (by virtue of being left adjoint to $(-)^Z$), it suffices to solve the following lifting problem for every representable $\mathsf{y}_{\mathbb{I}}(i)$:

$$\begin{tikzcd}
\mathsf{y}_{\mathbb{I}}(i) \times ¶ \arrow[r, tail] \arrow[d] & \mathsf{y}_{\mathbb{I}}(i) \times 1 \arrow[dl, dotted, "?"] \\
2
\end{tikzcd}$$

Observing that $\mathsf{y}_{\mathbb{I}}(0)$ is the proposition $¶$, we just need to check the case for $\mathsf{y}_{\mathbb{I}}(1) = 1_{\widehat{\mathbb{I}}}$, which reduces to an ordinary lifting problem as follows.

$$\begin{tikzcd}
¶ \arrow[r, tail] \arrow[d] & 1 \arrow[dl, dotted, "?"] \\
2
\end{tikzcd}$$

Unfolding this diagram in **Set**, we have the following situation:

$$\begin{tikzcd}
0 \arrow[r] \arrow[d] \arrow[rr, bend left, "!"] & 1 \arrow[r, dashed, "?"] \arrow[d, equal] & 2 \arrow[d, equal] \\
1 \arrow[r, equal] \arrow[rr, bend right] & 1 \arrow[r, dashed, "?"'] & 2
\end{tikzcd}$$

As input we are given a commuting outer square $\begin{array}{ccc} 0 & & 2 \\ \downarrow & \to & \downarrow \\ 1 & & 2 \end{array}$, but in this case this data consists of just the downstairs map $1 \to 2$. We must fill in the indicated square. Observing that both indicated maps must be equal, we immediately conclude that the given map $1 \to 2$ corresponds to a unique map (itself) on the bottom as indicated.

□ **(7.4∗9)** We have that 2 is a predomain in the sense of **(7.3.1∗6)**.

■ **(7.4∗10)** By definition we need to show that 2 is replete. We observe that 2 is isomorphic to its type of singletons:

$$2 \cong \left\{\phi : \Sigma^2 \mid (\forall[a,b:2]\ \phi\ a \wedge \phi\ b \to a = b) \wedge (\phi(\mathsf{inl} \cdot \star) \vee \phi(\mathsf{inr} \cdot \star))\right\}$$

Because Σ is closed under finite joins by **(7.4∗4)**, the type of singletons of 2 can be defined as the limit of a diagram of replete types, and so it is replete as well.

□ **(7.4∗11)** We have that $(\mathrm{Sh}(\mathscr{D}), \Sigma, \P)$ is a model of cost-sensitive synthetic domain theory in the sense of **(7.3.5∗2)**.

■ **(7.4∗12)** By **(7.4∗4)**, **(7.4∗5)** and **(7.4∗7)**.

CHAPTER 8

Cost-sensitive computational adequacy for PCF

(8∗1) In this chapter we extend the results of Chapter 5 to **PCF** by means of the cost-sensitive synthetic domain theory developed in Chapter 7. In the following we fix a model of cost-sensitive synthetic domain theory $\mathscr{E}$ with a dominance Σ and phase proposition ¶.

(8∗2) The language **PCF** was introduced in Plotkin [96] to study the relationship between denotational and operational semantics; *op. cit.* was also where the notion of computational adequacy was first introduced. It would not be an exaggeration to say that **PCF** is the progenitor of all programming languages equipped with higher-order recursion.

(8∗3) In contrast to Chapter 5, we work with a dynamic semantics of **PCF** not based on taking the reflexive-transitive closure of the one step transition $\mapsto$ but rather defined directly as a partial computation. This decision is necessitated by the proof of the computational adequacy property, which requires that the relation $e \Downarrow^c v$ coming from the one step transition relation as in **(5.1∗6)** is a Σ-proposition. A way to show that the reflexive-transitive closure of a decidable relation is a Σ-proposition is to close Σ under countable joins — a property that need not hold in general models of synthetic domain theory (and indeed unlikely to hold in the model in Section 7.4). The approach we take in this dissertation is to define the "big-step" semantics of a one step transition relation as a recursive function, which always exist in models of synthetic domain theory. In Chapter 9 we shall discuss the relationship between the two semantics in the context of internal *vs.* external adequacy.

8.1. COST-SENSITIVE PCF: $\mathbf{PCF}_{\mathsf{cost}}$

(8.1∗1) We have already discussed **PCF** in **(4.1.3∗8)** and **(6∗1)** and the representation of cost structure as an effect in **(4.1.3∗9)**; for completeness we record the syntax of this language, dubbed $\mathbf{PCF}_{\mathsf{cost}}$:

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{nat}} \qquad \frac{\Gamma \vdash v : \mathsf{nat}}{\Gamma \vdash \mathsf{suc}(v) : \mathsf{nat}}$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \qquad \Gamma \vdash e_0 : X \qquad \Gamma, z : \mathsf{nat} \vdash e_1 : X}{\Gamma \vdash \mathsf{ifz}(e, e_0, z.e_1) : X} \qquad \frac{\Gamma, x : A \vdash e : X}{\Gamma \vdash \lambda x.e : A \to X}$$

$$\frac{\Gamma \vdash e : A \to X \qquad \Gamma \vdash e_1 : A}{\Gamma \vdash e\ e_1 : X} \qquad \frac{\Gamma, x : \mathsf{U}X \vdash e : X}{\Gamma \vdash \mathsf{fix}(x.e) : X} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{ret}(a) : \mathsf{F}A}$$

$$\frac{\Gamma \vdash e : \mathsf{F}A \qquad \Gamma, a : A \vdash e_1 : X}{\Gamma \vdash \mathsf{bind}(e, a.e_1) : X} \qquad \frac{}{\Gamma \vdash \mathsf{yes} : 2} \qquad \frac{}{\Gamma \vdash \mathsf{no} : 2} \qquad \frac{}{\Gamma \vdash \star : 1} \qquad \frac{c : \mathbb{C} \qquad \Gamma \vdash e : X}{\Gamma \vdash \mathsf{step}^c(e) : X}$$

Observe that the syntax of $\mathbf{PCF}_{\mathsf{cost}}$ may be parameterized in an arbitrary (internal) monoid $(\mathbb{C}, +, 0)$. Semantically we require additional properties on $\mathbb{C}$, which we delineate in **(8.2∗1)**.

(8.1∗2) We will write $\mathsf{tm}(\Gamma, A)$ and $\mathsf{tm}(A)$ for the set of well-typed terms $\Gamma \vdash e : A$ and closed terms $e : A$, respectively.

8.2. DENOTATIONAL COST SEMANTICS

(8.2∗1) In keeping with the model of **calf** in Section 4.4, we parameterize the developments in this section with a restriction-modal monoid $(\mathbb{C}, 0, +)$ (we drop the preorder relation since it plays no role in the denotational semantics). In addition, because we intend to interpret the types of $\mathbf{PCF}_{\mathsf{cost}}$ into synthetic (pre)domains, we require $\mathbb{C}$ to be a predomain. Consequently the sealing modality $\bullet - = \P \vee -$ used to implement the phase distinction must be computed as a colimit in the category of synthetic predomains (which recall from **(3.5.1∗4)** need not coincide with the corresponding colimit computed in the ambient topos).

To facilitate the proof of the computational adequacy property, we require the cost monoid $(\mathbb{C}, +, 0)$ to be *computational* in the sense that $\mathbb{C}$ has Σ-equality. This property is used in two places: when reasoning about the *computational semantics* **(8.2.4∗2)** of $\mathbf{PCF}_{\mathsf{cost}}$, we need $\bullet\mathbb{C}$ to be discrete (which follows from **(7.3.6.2∗3)** and **(7.3.6.2∗6)**) in order to prove the property that sequential composition of computations may be decomposed **(8.2.4∗6)**; the discreteness of $\bullet\mathbb{C}$ is used again when showing that the *formal approximation predicates* **(8.4∗2)** associated to semantic domains are admissible in the sense of **(7.3.7.1∗1)**.

8.2.1. The partial cost monad.

(8.2.1∗1) We will define a model of $\mathbf{PCF}_{\mathsf{cost}}$ based on the algebra models of call-by-push-value, which we have already worked through in Section 4.4 for **calf** in the total case. To account for partiality, we work with the *partial cost monad* $\mathbb{T}$, whose underlying functor composes the lift monad with the writer monad for the sealed cost monoid: $\mathsf{T}(A) = \mathsf{L}(\bullet\mathbb{C} \times A)$. The monad structure on T arises by the following distributive law:

$$\tau : \bullet\mathbb{C} \times \mathsf{L}A \to \mathsf{L}(\bullet\mathbb{C} \times A)$$

$\tau(c, (\phi, f)) = (\phi, \lambda u : \phi.\ (c, fu))$

Explicitly we obtain the following monad structure:

$\eta_{\mathbb{T}}(a) = \eta_{\mathsf{L}}(0, a)$
$\mu_{\mathbb{T}}(e) = (c, x) \leftarrow_{\mathbb{L}} e; (c', a) \leftarrow_{\mathbb{L}} x; (c + c', a)$

In the above we write $x \leftarrow_{\mathbb{M}} e; f(x)$ for the induced bind operation of a monad $\mathbb{M}$ where $f : A \to \mathsf{M}(B)$ is a map into a free $\mathbb{M}$-algebra.

(8.2.1∗2) As in Section 4.4.3 we have a map $f^\sharp : \mathsf{T}A \to X$ for every map $f : A \to X$ into a $\mathbb{T}$-algebra satisfying $f^\sharp(\eta_{\mathsf{T}}(a)) = f(a)$.

8.2.2. The derived cost algebra.

(8.2.2∗1) To facilitate equational reasoning about the partial cost monad and its algebras, we rely on some general properties about monads associated to a distributive law. In this section we fix strong monads $\mathbb{M}_1, \mathbb{M}_2$ on a Cartesian closed category $\mathscr{C}$ such that the composite monad $\mathbb{M} = \mathbb{M}_2\mathbb{M}_1$ arising from a distributive law $l : \mathbb{M}_2\mathbb{M}_1 \to \mathbb{M}_1\mathbb{M}_2$ is also strong. We denote the strength maps by s, s_1, s_2 respectively.

☐ **(8.2.2∗2)** Given an $\mathbb{M}$-algebra $\alpha : MX \to X$, we have that $\boxplus_X : M_1X \xrightarrow{\eta_2} M_2(M_1X) \xrightarrow{\alpha} X$ is a $\mathbb{M}_1$-algebra and that $\boxtimes_X : M_2X \xrightarrow{M_2\eta_2} M_2(M_1X) \xrightarrow{\alpha} X$ is a $\mathbb{M}_2$-algebra [11, Section 2].

☐ **(8.2.2∗3)** We have that $\boxplus_{MA} = M_2\mu_1 \circ l : M_1(MA) \to MA$ holds for the free $\mathbb{M}_1$-algebra. Diagrammatically,

$$\begin{tikzcd}
M_1M_2 \arrow[r, "\eta_1"] \arrow[d, "l"'] & (M_2M_1)^2 \arrow[d, "\alpha_{MX} = \mu_M"] \\
M_2M_1^2 \arrow[r, "M_2\mu_1"'] & M_2M_1
\end{tikzcd}$$

■ **(8.2.2∗4)** The result follows from a diagram chase of naturality squares:

$$\begin{tikzcd}
M_1M_2M_1 \arrow[r, "\eta_2"] \arrow[d, "l"'] & M_2M_1M_2M_1 \arrow[d, "M_2l"] \\
M_2M_1^2 \arrow[r, "\eta_2"] \arrow[d, "M_2\mu_1"'] & M_2^2M_1^2 \arrow[d, "M_2^2\mu_1"] \\
M_2M_1 \arrow[r, "\eta_2"] \arrow[dr, equal] & M_2^2M_1 \arrow[d, "\mu_2"] \\
 & M_2M_1
\end{tikzcd}$$

□ **(8.2.2∗5)** For every map $f : A \to |X|$ where $\alpha : M|X| \to |X|$ is an $\mathbb{M}$-algebra, we have that $f^\sharp \boxplus_{MA} = \boxplus_{MA} M_1 f^\sharp$:

$$\begin{tikzcd}
M_1M_2M_1A \arrow[r, "\boxplus_{MA}"] \arrow[d, "f^\sharp"'] & M_2M_1A \arrow[d, "M_1f^\sharp"] \\
M_1|X| \arrow[r, "\boxplus_X"'] & |X|
\end{tikzcd}$$

■ **(8.2.2∗6)** The proof again follows by expanding definitions and diagram chasing:

$$\begin{tikzcd}
M_1M_2M_1A \arrow[r, "\eta_2"] \arrow[d, "M_1M_2M_1f"'] & M_2M_1M_2M_1A \arrow[r, "\boxplus_{MA} = \mu"] \arrow[d, "M^2f"] & M_2M_1A \arrow[d, "Mf"] \\
M_1M_2M_1|X| \arrow[r, "\eta_2"'] \arrow[d, "M_1\alpha"'] & M_2M_1M_2M_1|X| \arrow[r, "\boxplus_{MA} = \mu"'] \arrow[d, "M\alpha"] & M_2M_1|X| \arrow[d, "\alpha"] \\
M_1|X| \arrow[r, "\eta_2"'] & M_2M_1|X| \arrow[r, "\alpha"'] & |X|
\end{tikzcd}$$

The bottom right square commutes by the coherence law for algebras and the other three inner squares are naturality squares.

□ **(8.2.2∗7)** Recall from Levy [72] that given a strong monad $\mathbb{M}$ with a strength s and a $\mathbb{M}$-algebra $\alpha : M|X| \to |X|$, there is a unique $\mathbb{M}$-algebra structure on $A \to |X|$ whose structure map $\alpha_{A\to|X|}$ is defined as the exponential transpose of the following maps:

$$M(|X|^A) \times A \xrightarrow{s} M(|X|^A \times A) \xrightarrow{M\mathsf{ev}} M(|X|) \xrightarrow{\alpha} |X|$$

satisfying the following:

$$\begin{tikzcd}
M(|X|^A \times A) \arrow[r, "\alpha_{A\to|X|}"] \arrow[d, "M\mathsf{ev}"'] & {|X|^A \times A} \arrow[d, "\mathsf{ev}"] \\
M|X| \arrow[r, "\alpha_{|X|}"'] & {|X|}
\end{tikzcd}$$

︾ **(8.2.2∗8)** Given an $\mathbb{M}$-algebra X, we have a $\mathbb{M}_1$-algebra on $A \to |X|$. Then we may compute that $\mathsf{ev} \circ (\boxplus_{A\to X} \times A) = \boxplus_X \circ M_1\mathsf{ev} \circ s_1$ for the $\mathbb{M}_1$-algebra $A \to X$. Diagrammatically,

$$\begin{tikzcd}
M_1(|X|^A \times A) \arrow[r, "\boxplus_{A\to X} \times A"] \arrow[d, "s_1"'] \arrow[drr, "\boxplus_{A\to|X|}"] & {|X|^A \times A} \arrow[dr, "\mathsf{ev}"] & \\
M_1|X| \arrow[r, "M_1\mathsf{ev}"'] & {|X|} \arrow[r, "\boxplus_X"'] & {|X|}
\end{tikzcd}$$

︾ **(8.2.2∗9)** The action of the derived cost algebra satisfies the following equations for $e : \mathsf{T}(A)$ and $f : A \to X$ for some $\mathbb{T}$-algebra X:

$$\begin{aligned}
&c \boxplus_{\mathsf{T}(A)} e = (e{\downarrow}, \lambda u.\ \mu_{\bullet\mathbb{C}\times -}(c, e)) \\
&f^\sharp(c \boxplus_{\mathsf{T}(A)} e) = c \boxplus_X (f^\sharp\ e) \\
&(c \boxplus_{A\to X} f)\ a = c \boxplus_X (f\ a)
\end{aligned}$$

■ **(8.2.2∗10)** By **(8.2.2∗3)**, **(8.2.2∗5)** and **(8.2.2∗8)**.

8.2.3. Model of $\mathbf{PCF}_{\mathsf{cost}}$.

(8.2.3∗1) We interpret a value type as a predomain and a computation type as a predomain equipped with a $\mathbb{T}$-algebra structure. The interpretation of the basic call-by-push-value structure of $\mathbf{PCF}_{\mathsf{cost}}$ follows from the semantic adjunction between partial cost algebras and general predomains. We interpret types as follows.

$$\begin{aligned}
&[\![-]\!] : \mathsf{tp}^+ \to \mathcal{U}_{\mathsf{predom}} \\
&[\![-]\!] : \mathsf{tp}^- \to \mathsf{Alg}_{\mathbb{T}}(\mathcal{U}_{\mathsf{predom}})
\end{aligned}$$

$$\begin{aligned}
&[\![\mathsf{F}A]\!] = \mathsf{T}([\![A]\!]) \\
&[\![\mathsf{U}X]\!] = U([\![X]\!]) \\
&[\![1]\!] = 1 \\
&[\![2]\!] = 2 \\
&[\![\mathsf{nat}]\!] = \mathbb{N}_{\mathsf{P}}
\end{aligned}$$

$[\![A \to X]\!] = [\![A]\!] \to [\![X]\!]$

Observe that we interpret nat as $\mathbb{N}_\mathsf{P}$, the nno of predomains, and that $[\![A]\!] \to [\![X]\!]$ has a canonically given algebra structure by virtue of **(8.2.2∗7)**.

(8.2.3∗2) The types 1, 2, nat, and $A \to X$ have standard interpretations. We focus on the interpretation of the free algebra:

$$[\![-]\!] : \{\Gamma, A\}\ (\Gamma \vdash A) \to [\![\Gamma]\!] \to [\![A]\!]$$
$$[\![-]\!] : \{\Gamma, A\}\ (\Gamma \vdash X) \to [\![\Gamma]\!] \to U([\![X]\!])$$

$$[\![\mathsf{ret}(a)]\!](\gamma) = \eta_{\mathbb{T}}([\![a]\!](\gamma))$$
$$[\![\mathsf{step}(c, e)]\!](\gamma) = (\eta_\bullet c) \boxplus [\![e]\!](\gamma)$$
$$[\![\mathsf{bind}(e, f)]\!](\gamma) = [\![f]\!](\gamma)^\sharp([\![e]\!](\gamma))$$
$$[\![\mathsf{fix}(f)]\!](\gamma) = \mathsf{fix}(\lambda x.\ [\![f]\!](x, \gamma))$$

Observe that because every $\mathbb{T}$-algebra is also an $\mathbb{L}$-algebra by **(8.2.2∗2)** and thus a domain, we have that every semantic map $[\![X]\!] \to [\![X]\!]$ for a computation type possesses a fixed point by **(7.3.7∗5)**.

8.2.4. Computational semantics of PCF$_\mathsf{cost}$.

(8.2.4∗1) We begin with a family of small-step transition relations $\mapsto_A \subseteq \mathsf{tm}(A) \times \mathbb{C} \times \mathsf{tm}(A)$ that implements the *cost effect* model in the sense of Hoffmann [56] (congruence rules omitted):

$$\frac{}{\mathsf{bind}(\mathsf{ret}(a), f) \mapsto 0, f(a)} \qquad \frac{}{(\lambda x.e)\ e_1 \mapsto 0, e[e_1/x]} \qquad \frac{}{\mathsf{fix}(\lambda x.e) \mapsto 0, e[\mathsf{fix}(e)/x]}$$

$$\frac{}{\mathsf{ifz}(\mathsf{zero}, e_0, e_1) \mapsto 0, e_0} \qquad \frac{}{\mathsf{ifz}(\mathsf{succ}(v), e_0, e_1) \mapsto 0, e_1(v)} \qquad \frac{}{\mathsf{step}^c(e) \mapsto c, e}$$

The intuitive meaning of $e \mapsto c, e'$ is that e transitions in one step to e' and incurs cost c; the only place where cost is effected is at $\mathsf{step}^c(e) \mapsto c, e$. Because $\mapsto_A$ is decidable, we have a characteristic map $\mathsf{out} : \{A : \mathsf{tp}^+\}\ .\ \mathsf{tm}(A) \to 1 + (\mathbb{C} \times \mathsf{tm}(A))$.

(8.2.4∗2) To obtain a big-step cost semantics, we iterate the one step relation $\mapsto$ to obtain a partial map implementing the *computational semantics.* For any **PCF**$_\mathsf{cost}$ type $A : \mathsf{tp}^+$, we may define the following functional of type $(\mathsf{tm}(A) \times \mathsf{tm}(A) \to \mathsf{T}(1)) \to (\mathsf{tm}(A) \times \mathsf{tm}(A) \to \mathsf{T}(1))$.

$$\Phi_\mathsf{eval}\ f\ (e, v) = \begin{cases} (\eta_\bullet c) \boxplus f(e', v) & \mathsf{out}(e) = \mathsf{inr} \cdot (c, e') \\ (e = v, \lambda - .\ \eta_\bullet 0) & \mathsf{out}(e) = \mathsf{inl} \cdot \star \end{cases}$$

Recall that $\boxplus : \bullet\mathbb{C} \times \mathsf{T}(1) \to \mathsf{T}(1)$ is the induced cost algebra map on free $\mathbb{T}$-algebras **(8.2.2∗9)**. Define $\mathsf{eval} : \{A : \mathsf{tp}^+\}\ \mathsf{tm}(A) \times \mathsf{tm}(A) \to \mathsf{T}(1)$ to be the fixed-point of Φ_eval and $\mathsf{profile} : \mathsf{tm}^+(\mathsf{UF1}) \to \mathsf{T}(1)$ as $\mathsf{profile}(e) = \mathsf{eval}(e, \mathsf{ret}(\star))$. The meaning of $\mathsf{eval}_A(e, v) : \mathsf{T}(1)$ is that when it is defined, e computes to a value v incurring the defined cost. Similarly, $\mathsf{profile}(e)$ is the cost of computing e when the former is defined. In the following, we will establish some expected properties of the computational semantics such as the uniqueness of evaluation and a "big-step" law for sequencing evaluations.

☐ **(8.2.4∗3)** The relation $\pi_1 \circ \mathsf{eval}$ is functional, *i.e.* $\downarrow\mathsf{eval}(e, v)$ and $\downarrow\mathsf{eval}(e, v')$ implies $v = v'$.

Proof. Consider the following subset of $\Pi_{A:\mathsf{tp}}.\ \mathsf{tm}(\mathsf{UF}A) \to \mathsf{tm}(\mathsf{UF}A) \to \mathsf{T1}$:

$$P = \{\alpha \mid \forall[e, v, v']\ \alpha(e, v)\!\downarrow \wedge\ \alpha(e, v')\!\downarrow \to v = v'\}$$

By **(7.3.7.1∗2)** and **(7.3.7.1∗4)** P is admissible and proceed by fixed-point induction. Suppose that $\alpha \in P$ and that $\Phi_{\mathsf{eval}}(\alpha)(e, v)\!\downarrow$ and $\Phi_{\mathsf{eval}}(\alpha)(e, v')\!\downarrow$. We need to show that $v = v'$. We proceed by cases on $\mathsf{out}(e)$.

1. If $e \mapsto c', e'$, we may deduce that $\alpha(e', v)\!\downarrow$ and $\alpha(e', v')\!\downarrow$, so the result follows from the assumption that $\alpha \in P$.

2. Otherwise, we have that $e = v$ and $e = v'$ by definition of Φ_{eval}, and so $v = v'$.

□

☐ **(8.2.4∗4)** If $\mathsf{eval}(e, \mathsf{ret}(v)) = c_1$ and $\mathsf{eval}(g\ v, \mathsf{ret}(w)) = c_2$, then $\mathsf{eval}(e; g, \mathsf{ret}(w)) = c_1 + c_2$.

Proof. Consider the following subset of $\Pi_{A:\mathsf{tp}}.\ \mathsf{tm}(\mathsf{UF}A) \to \mathsf{tm}(\mathsf{UF}A) \to \mathsf{T1}$:

$$P = \{\alpha \mid \forall[e]\ \alpha(e, v)\!\downarrow \wedge\ \mathsf{eval}(g\ v, \mathsf{ret}(w))\!\downarrow \to \mathsf{eval}(e; g, \mathsf{ret}(w)) = \alpha(e, v) + \mathsf{eval}(g\ v, \mathsf{ret}(w))\}$$

It suffices to show that $\mathsf{eval} \in P$. Observing that P is admissible, we proceed by fixed-point induction. Suppose that $\alpha \in P$, $\Phi_{\mathsf{eval}}(\alpha)(e, v)\!\downarrow$, and that $\mathsf{eval}(g\ v, \mathsf{ret}(w))\!\downarrow$. We need to show that $\mathsf{eval}(e; g, \mathsf{ret}(w)) = \Phi_{\mathsf{eval}}(\alpha)(e, v) + \mathsf{eval}(g\ v, \mathsf{ret}(w))$. We proceed by cases on $\mathsf{out}(e)$.

1. If $e \mapsto c', e'$, then we compute:

$$\begin{aligned}\mathsf{eval}(e; g, \mathsf{ret}(w)) &= c' \boxplus \mathsf{eval}(e'; g, \mathsf{ret}(w))\\ &= c' + \alpha(e', v) + \mathsf{eval}(g\ v, \mathsf{ret}(w))\\ &= \Phi_{\mathsf{eval}}(\alpha)(e, v) + \mathsf{eval}(g\ v, \mathsf{ret}(w))\end{aligned}$$

   Where the first equality follows from the assumption that $\alpha \in P$ and the second by the definition of Φ_{eval}.

2. Otherwise, we have that e val. Since $\Phi_{\mathsf{eval}}(\alpha)(e, \mathsf{ret}(v))\!\downarrow = (e = \mathsf{ret}(v), 0)\!\downarrow = (e = \mathsf{ret}(v))$ holds, we can compute:

$$\begin{aligned}\mathsf{eval}(e; g, \mathsf{ret}(w)) &= \mathsf{eval}(\mathsf{ret}(v); g, \mathsf{ret}(w))\\ &= \Phi_{\mathsf{eval}}(\mathsf{eval})(\mathsf{ret}(v); g, \mathsf{ret}(w))\\ &= \mathsf{eval}(g\ v, \mathsf{ret}(w))\end{aligned}$$

   But this is what we needed to show since $\Phi_{\mathsf{eval}}(\alpha)(e, \mathsf{ret}(v)) = 0$.

□

» **(8.2.4∗5)** If $\mathsf{eval}(e, \mathsf{ret}(v)) = c_1$ and $\mathsf{profile}(g\ v) = c_2$, then $\mathsf{profile}(e; g) = c_1 + c_2$.

☐ **(8.2.4∗6)** The following inference rule is valid for any Σ-predicate φ:

$$\frac{\forall[v : A]\ \mathsf{eval}(e, v)\!\downarrow \wedge\ \mathsf{eval}(g\ v, \mathsf{ret}(w))\!\downarrow \to \varphi(\mathsf{eval}(e, v) + \mathsf{eval}(g\ v, \mathsf{ret}(w)))}{\mathsf{eval}(e; g, \mathsf{ret}(w))\!\downarrow \to \varphi(\mathsf{eval}(e; g, \mathsf{ret}(w)))}$$

■ **(8.2.4∗7)** Consider the subset P defined as the intersection of the following subsets:

$$Q = \{\alpha \mid \alpha \sqsubseteq \mathsf{eval}\}$$
$$R = \{\alpha \mid \forall[c', e', n]\ (e \mapsto^n c', e') \wedge (\alpha(e'; g, \mathsf{ret}(w))\downarrow) \rightarrow \varphi(c' + \alpha(e'; g, \mathsf{ret}(w)))\}$$

It suffices to show that $\mathsf{eval} \in P$. We have that P is admissible, and we proceed by fixed-point induction. Suppose that $\alpha \in P$. We need to show that $\Phi_{\mathsf{eval}}(\alpha) \in P$, where Φ_{eval} is the characteristic functional of eval (Section 8.2.4). It's immediate that $\Phi_{\mathsf{eval}}(\alpha) \in Q$. It remains to show that it is also contained in R. So suppose that $e \mapsto^n c', e'$ and $\Phi_{\mathsf{eval}}(\alpha)(e'; g, \mathsf{ret}(w))\downarrow$. We want to show that $\varphi(c' + \Phi_{\mathsf{eval}}(\alpha)(e'; g, \mathsf{ret}(w)))$. We proceed by cases on $\mathsf{out}(e')$.

1. If $\mathsf{out}(e') = \mathsf{inl} \cdot \star$, then we know that $e' = \mathsf{ret}(v)$ for some $v : \mathsf{UF1}$. Stepping the operational semantics, we have that $\mathsf{ret}(v); g \mapsto 0, g\ v$, and by definition of the computational semantics $\Phi_{\mathsf{eval}}(\alpha)(e'; g, \mathsf{ret}(w)) = 0 \boxplus \alpha(g\ v, \mathsf{ret}(w)) = \alpha(g\ v, \mathsf{ret}(w))$. Since we assumed $\alpha \in Q \iff \alpha \sqsubseteq \mathsf{eval}$, we also have $\mathsf{eval}(g\ v, \mathsf{ret}(w))\downarrow$, and since $\mathbb{C}$ is discrete by **(8.2∗1)** so is $\bullet\mathbb{C}$ by **(7.3.6.2∗6)**, from which it follows that $\bullet\mathbb{C}$ is discrete by **(7.3.6.2∗3)**. Thus we have $\alpha(g\ v, \mathsf{ret}(w)) = \mathsf{eval}(g\ v, \mathsf{ret}(w))$. Recalling the premise and the fact that $\mathsf{eval}(e, \mathsf{ret}(v)) = c'$, we may conclude that $\varphi(c' + \mathsf{eval}(g\ v, \mathsf{ret}(w)))$, which is what we needed to show.

2. Otherwise, $\mathsf{out}(e') = \mathsf{inr} \cdot (c'', e'')$ for some $e'' : \mathsf{UF1}$, and we have that $e'; g \mapsto c'', e''; g$. By definition of the computational semantics, this means that $\Phi_{\mathsf{eval}}(\alpha)(e'; g, \mathsf{ret}(w)) = c'' \boxplus \alpha(e''; g, \mathsf{ret}(w))$. Since we assumed that $c'' \boxplus \alpha(e''; g, \mathsf{ret}(w))\downarrow$, we can use the laws of the derived algebra (**(8.2.2∗9)**) to deduce that $\alpha(e''; g, \mathsf{ret}(w))\downarrow$ as well, and so by the assumption that $\alpha \in R \subseteq P$, we have that $\varphi(c' + c'' + \alpha(e''; g, \mathsf{ret}(w)))$ holds, which is what we needed to show.

□ **(8.2.4∗8)** We have $\mathsf{profile}((e; g); i) = \mathsf{profile}(e; (\lambda v.\ g\ v; i))$.

■ **(8.2.4∗9)** In one direction, we show that $\mathsf{profile}((e; g); i)\downarrow$ implies $\mathsf{profile}(e; (\lambda v.\ g\ v; i))\downarrow$ and both denote identical costs. Consider the Σ-predicate φ such that $\varphi(c)$ if and only if $\mathsf{profile}(e; (\lambda v.\ g\ v; h)) = c$. Suppose that $\mathsf{eval}(e; g, \mathsf{ret}(w))\downarrow$ and $\mathsf{profile}(i\ w)\downarrow$. By computational induction on sequencing **(8.2.4∗6)**, it suffices to show that $\mathsf{profile}(e; (\lambda v.\ g\ v; h)) = \mathsf{eval}(e; g, \mathsf{ret}(w)) + \mathsf{profile}(i\ w)$. Applying computational induction on $\mathsf{eval}(e; g, \mathsf{ret}(w))\downarrow$, we further suppose that $\mathsf{eval}(e, \mathsf{ret}(v))\downarrow$ and $\mathsf{eval}(g\ v, \mathsf{ret}(w))\downarrow$ and aim to show that $\mathsf{profile}(e; (\lambda v.\ g\ v; h)) = \mathsf{eval}(e, \mathsf{ret}(v)) + \mathsf{eval}(g\ v, \mathsf{ret}(w)) + \mathsf{profile}(i\ w)$.

1. We claim that $\mathsf{profile}(e; (\lambda v.\ g\ v; i))\downarrow$. By the big-step semantics of profiling **(8.2.4∗5)**, it suffices to show that $\mathsf{eval}(e, \mathsf{ret}(v))\downarrow$ for some v and $\mathsf{profile}(g\ v; i)\downarrow$. The former follows from our assumption; for the latter, it suffices to show that $\mathsf{eval}(g\ v, \mathsf{ret}(w))\downarrow$ and $\mathsf{profile}(i\ w)\downarrow$, both of which follow from assumptions.

2. Given that $\mathsf{profile}(e; (\lambda v.\ g\ v; i))\downarrow$, we may apply computational induction again: supposing that $\mathsf{eval}(e, \mathsf{ret}(v'))\downarrow$ and $\mathsf{profile}(g\ v; i)\downarrow$, we have to show that $\mathsf{eval}(e, \mathsf{ret}(v')) + \mathsf{profile}(g\ v; i) = \mathsf{eval}(e, \mathsf{ret}(v)) + \mathsf{eval}(g\ v, \mathsf{ret}(w)) + \mathsf{profile}(i\ w)$, which follows from the uniqueness of evaluation **(8.2.4∗3)** and big-step semantics of profiling **(8.2.4∗5)**.

In the other direction, suppose that $\mathsf{profile}(e; (\lambda v.\ g\ v; i))\downarrow$. It suffices to show that $\mathsf{profile}((e; g); i)\downarrow$. By computational induction, we may assume that $\mathsf{eval}(e, \mathsf{ret}(v))\downarrow$ and $\mathsf{profile}(g\ v; i)\downarrow$. Applying computational induction again, we can also assume that $\mathsf{eval}(g\ v, \mathsf{ret}(w))\downarrow$ and $\mathsf{profile}(i\ w)\downarrow$ for some w. By the big-step semantics of profiling **(8.2.4∗5)**, it suffices to show that $\mathsf{eval}(e; g, \mathsf{ret}(w))\downarrow$ and $\mathsf{profile}(i\ w)\downarrow$. The latter is our assumption, and the former follows from the big-step semantics of evaluation **(8.2.4∗4)**.

□ **(8.2.4∗10)** The following is valid:

$$\frac{\forall[e]\ \mathsf{eval}(f, \lambda e)\downarrow \wedge \mathsf{eval}(e[v], \mathsf{ret}(w))\downarrow \to \varphi(\mathsf{eval}(f, \lambda e) + \mathsf{eval}(e[v], \mathsf{ret}(w)))}{\mathsf{eval}(f\ v, \mathsf{ret}(w))\downarrow \to \varphi(\mathsf{eval}(f\ v, \mathsf{ret}(w)))}$$

■ **(8.2.4∗11)** Similar to **(8.2.4∗6)**.

□ **(8.2.4∗12)** We have that $\mathsf{eval}((e; g)\ w, z) = \mathsf{eval}(e; \lambda v.\ g\ v\ w, z)$.

■ **(8.2.4∗13)** Similar to **(8.2.4∗8)**.

8.3. SOUNDNESS

(8.3∗1) In this section we show that the denotational semantics is *sound*, which means that the computational steps are respected by the denotational semantics:

□ **(8.3∗2)** If $e \mapsto c, e'$, then $[\![e]\!] = c \boxplus [\![e']\!]$.

■ **(8.3∗3)** By induction on the derivation of $e \mapsto c, e'$.

□ **(8.3∗4)** If $\mathsf{eval}(e, v)\downarrow$, then $[\![e]\!] = \mathsf{eval}(e, v) \boxplus [\![v]\!]$.

(8.3∗5) Consider the following subset:

$$P = \{\alpha \mid \forall[e]\ \alpha(e, v)\downarrow \to [\![e]\!] = \mathsf{eval}(e, v) \boxplus [\![v]\!]\}$$

Because $[\![e]\!] = \mathsf{eval}(e, v) \boxplus [\![v]\!]$ is a Σ-proposition, we see that P is an admissible subset. Suppose that $\alpha \in P$ and $\Phi_{\mathsf{eval}}(\alpha)(e, v)\downarrow$. We need to show that $[\![e]\!] = \mathsf{eval}(e, v) \boxplus [\![v]\!]$. We proceed by cases on $\mathsf{out}(e)$.

1. If $e \mapsto c', e'$, then by the soundness of the one step relation **(8.3∗2)**, it suffices to show that $c' \boxplus [\![e']\!] = c' \boxplus \mathsf{eval}(e', v) \boxplus [\![v]\!]$, which follows from the assumption, noting that $\Phi_{\mathsf{eval}}(\alpha)(e, v)\downarrow$ implies $\alpha(e, v)\downarrow$.

2. Otherwise, we have that $e = v$, and so the result holds since $\mathsf{eval}(e, e) = 0$.

⌄ **(8.3∗6)** Given $e : \mathsf{UF1}$, we have $\mathsf{profile}(e) \sqsubseteq [\![e]\!]$.

□ **(8.3∗7)** As a simple corollary of the soundness we obtain a rigorous proof of the intuitive fact that computations may not observe the cost effect: any $e : \mathsf{UF1} \rightharpoonup \mathsf{F2}$ is weakly, functionally constant in the sense that for all $x, y : \mathsf{UF1}$, if $\mathsf{profile}(x)\downarrow$ and $\mathsf{profile}(y)\downarrow$, then $\mathsf{eval}(e\ x, \mathsf{ret}(v))\downarrow$ and $\mathsf{eval}(e\ y, \mathsf{ret}(u))\downarrow$ imply $v = u$.

■ **(8.3∗8)** Let $c : \bullet\mathbb{C}$ and $d : \bullet\mathbb{C}$ be the costs denoted by $\mathsf{eval}(e\ x, \mathsf{ret}(v))$ and $\mathsf{eval}(e\ y, \mathsf{ret}(u))$, and c_x and $c_y : \bullet\mathbb{C}$ be the costs denoted by $\mathsf{profile}(x)$ and $\mathsf{profile}(y)$. By soundness **(8.3∗4)** and laws of the derived algebra **(8.2.2∗9)**, we have that $[\![e\ x]\!] = c \boxplus \eta_{\mathbb{T}}([\![v]\!]) = \eta_{\mathbb{L}}(c, [\![v]\!])$ and similarly $[\![e\ y]\!] = d \boxplus \eta_{\mathbb{T}}([\![u]\!]) = \eta_{\mathbb{L}}(d, [\![u]\!])$. It suffices to show that $[\![v]\!] =_2 [\![u]\!]$. Because 2 is a restriction-modal type by **(7.3.5∗2)**, we may assume that ¶ holds. On the other hand, by **(8.3∗6)**, we have $c_x \sqsubseteq [\![x]\!]$ and $c_y \sqsubseteq [\![y]\!]$. By **(7.3.6.1∗5)** and the fact that $\bullet\mathbb{C}$ is sealed, this means that $[\![x]\!] = [\![y]\!]$ as they are both elements of 1. Thus we have $[\![e\ x]\!] = \eta_{\mathbb{L}}(c, [\![v]\!]) = \eta_{\mathbb{L}}(d, [\![u]\!]) = [\![e\ y]\!]$, which means that $[\![v]\!] = [\![u]\!]$.

8.4. COMPUTATIONAL ADEQUACY

(8.4∗1) In this section we prove the converse to **(8.3∗4)** at base type: denotational steps are respected by the computational semantics. Similar to Section 5.2.4, we employ a binary logical relation argument between the semantics and syntax of $\mathbf{PCF}_{\mathsf{cost}}$.

(8.4∗2) The *formal approximation relations* for $\mathbf{PCF}_{\mathsf{cost}}$ are a family of relations $\lhd_A \subseteq [\![A]\!] \times \mathsf{tm}(A)$ indexed in value types A of $\mathbf{PCF}_{\mathsf{cost}}$ defined by recursion on A as follows.

$$
\begin{aligned}
&e \lhd_1 e' = \top \\
&e \lhd_2 e' = (e = [\![e']\!]) \\
&e \lhd_{\mathsf{nat}} e' = (e = [\![e']\!]) \\
&\mathsf{adq}(e, e') = (e \sqsubseteq \mathsf{profile}(e')) \\
&e\ (R \Rightarrow S)\ e' = \forall[a\ R\ a']\ (e\ a)\ S\ (e'\ a') \\
&e \lhd_{\mathsf{UF}A} e' = \forall[f\ (\lhd_A \Rightarrow \mathsf{adq})\ f']\ \mathsf{adq}(f^\sharp(e), e'; f') \\
&e \lhd_{\mathsf{U}(A \to X)} e' = e\ (\lhd_A \Rightarrow \lhd_{\mathsf{U}X})\ e'
\end{aligned}
$$

The meaning of the logical relations interpretation of free algebras is as follows: $e : [\![\mathsf{UF}A]\!]$ is related to $e' : \mathsf{tm}(\mathsf{UF}A)$ whenever given a pair of logically related continuations $f\ (\lhd_A \Rightarrow \mathsf{adq})\ f'$ at type $\mathsf{UF}1$, the corresponding semantic and syntactic sequences $f^\sharp(e)$ and $e'; f'$ are also related by the relation adq, which is precisely the adequacy property we want to prove.

(8.4∗3) Formal approximation relations may be extended to contexts in the evident way. We write $\Gamma \vdash e \lhd_A e'$ when for all closing substitutions $s \lhd_\Gamma \sigma$, we have that $e(s) \lhd_A e'[\sigma]$ holds.

(8.4∗4) As for the case for **STLC** in Section 5.2.5, the computational adequacy property follows by proving the fundamental lemma of logical relations. In order to apply fixed-point induction in the fixed-point case, we must first show that all subsets of the form $\lhd_{\mathsf{U}X} e$ for a computation $e : \mathsf{tm}(\mathsf{U}X)$ is admissible.

8.4.1. Admissibility properties.

☐ **(8.4.1∗1)** We have that $- \lhd_{\mathsf{UF}A}\ e$ is an admissible subset of $\mathsf{T}(A)$.

(8.4.1∗2) By **(7.3.7.1∗10)** it suffices to show downward closure and closure under $\bot$ and synthetic ω-joins.

1. By definition this means to show $f^\sharp(\bot) \sqsubseteq \mathsf{profile}(e; g)$ for all $f\ (A \Rightarrow \mathsf{adq})\ g$. But this holds since $f^\sharp(\bot) = \bot$ and $\bot$ is the least element of $\mathsf{T}(A)$.

2. Let d be a synthetic ω-chain such that $d_i \lhd_{\mathsf{UF}A} e$ for all $i : \omega$. We want to show that $\bigvee d \lhd_{\mathsf{UF}A} e$, which is to show that $f^\sharp(\bigvee d) \sqsubseteq \mathsf{profile}(e; g)$ for all $f\ (A \Rightarrow \mathsf{adq})\ g$. Since $f^\sharp(\bigvee d) = \bigvee(f^\sharp \circ d)$, this means to show $\bigvee(f^\sharp \circ d) \sqsubseteq \mathsf{profile}(e; g)$. By the universal property of the synthetic ω-join, it suffices to show $f^\sharp(d_i) \sqsubseteq \mathsf{profile}(e; g)$ for all $i : \omega$, but this is the assumption.

3. Fixing $d' \sqsubseteq d \lhd_{\mathsf{UF}A} e$, we need to show that $d' \lhd_{\mathsf{UF}A} e$. Suppose that $f\ (A \Rightarrow \mathsf{adq})\ g$. We need to show $f^\sharp(d') \sqsubseteq \mathsf{profile}(e;g)$. By the characterization of the order on lifts **(7.3.6.1∗5)**, we suppose $f^\sharp(d')\downarrow$ and show that $\mathsf{profile}(e;g)\downarrow$ and that $f^\sharp(d') = \mathsf{profile}(e;g)$. By assumption we know $d' = \eta_{\mathbb{L}}(a')$ and $f(a') = c$ for some $a : [\![A]\!]$ and $c : \bullet\mathbb{C}$. Since $d' \sqsubseteq d$, we know $d' = \eta_{\mathbb{L}}(a)$ for some a such that $a' \sqsubseteq a$. Consequently, we have $c = f(a') \sqsubseteq f(a)$, but since $\bullet\mathbb{C}$ is discrete (by **(8.2∗1)** and the argument in **(8.2.4∗7)**), we have $f(a) = c = f(a')$. Thus by the assumption that $d \lhd_{\mathsf{UF}A} e$, we have $f^\sharp(d) = c \sqsubseteq \mathsf{profile}(e;g)$. Again by the discreteness of $\bullet\mathbb{C}$ we have that $f^\sharp(d') = f(a') = c = \mathsf{profile}(e;g)$, as required.

☐ **(8.4.1∗3)** If $- \lhd_{\mathsf{U}X} e$ is admissible for all $e : \mathsf{U}X$, then $- \lhd_{\mathsf{U}(A\to X)} e$ is admissible for all $e : \mathsf{U}(A \to X)$.

■ **(8.4.1∗4)** Again we show downward closure and closure under $\bot$ and $\bigvee$.

1. Because $\bot(a) = \bot$, we have that $\bot \lhd_{\mathsf{U}(A\to X)} e$ by the assumption that $\bot \lhd_{\mathsf{U}X} e$ for all e.
2. Suppose that $f_i \lhd_{\mathsf{U}(A\to X)} e$. We need to show that $\bigvee f \lhd_{\mathsf{U}(A\to X)} e$. Suppose that $a \lhd_A b$. We need to show that $(\bigvee f)\ a \lhd_{\mathsf{U}X} e\ b$. This follows the fact that synthetic ω-joins in function spaces are computed pointwise and the assumption that $- \lhd_{\mathsf{U}X} e\ b$ is closed under $\bigvee$.
3. Fix $f' \sqsubseteq f \lhd_{\mathsf{U}(A\to X)} e$. To show that $f' \lhd_{\mathsf{U}(A\to X)} e$, suppose that $a \lhd_A b$. We need to show that $f'\ a \lhd_{\mathsf{U}X} e\ b$. By the premise, we have that $- \lhd_{\mathsf{U}X} e\ b$ is a lower set, so it suffices to show $f'\ a \sqsubseteq f\ a \lhd_{\mathsf{U}X} e\ b$, which follow from the assumptions $f' \sqsubseteq f$ and $f \lhd_{\mathsf{U}(A\to X)} e$.

☐ **(8.4.1∗5)** Given $e : \mathsf{U}X$, we have that $- \lhd_{\mathsf{U}X} e$ is an admissible subset of $[\![\mathsf{U}X]\!]$.

■ **(8.4.1∗6)** By **(8.4.1∗1)** and **(8.4.1∗3)**.

8.4.2. Fundamental lemma.

(8.4.2∗1) In this section we prove the fundamental lemma of logical relations for the formal approximation relations. We give the representative cases of the proof by induction on the derivation of terms.

☐ **(8.4.2∗2)** If $d \lhd_X e$ and $e' \mapsto c, e$, then $c \boxplus d \lhd_X e'$.

■ **(8.4.2∗3)** By induction on X, using the laws of the cost algebra **(8.2.2∗9)**.

☐ **(8.4.2∗4)** If $a \lhd_A v$, then $\eta_{\mathbb{T}}(a) \lhd_{\mathsf{UF}A} \mathsf{ret}(v)$.

■ **(8.4.2∗5)** Let $f\ (\lhd_A \Rightarrow \mathsf{adq})\ g$. We need to show that $(f^\sharp(\eta_{\mathbb{T}}(a)))\ \mathsf{adq}\ (\mathsf{ret}(v);g)$. Computing the denotational semantics and applying **(8.4.2∗2)**, it suffices to show that $(f\ a)\ \mathsf{adq}\ (g\ v)$, which follows from our assumption.

☐ **(8.4.2∗6)** If $d \lhd_{\mathsf{UF}A} e$ and $f \lhd_{\mathsf{U}(A\to X)} g$, then $f^\sharp(d) \lhd_{\mathsf{U}X} e;g$.

■ **(8.4.2∗7)** By induction on X.

1. If $X = \mathsf{F}B$, let $h\ (\lhd_B \Rightarrow \mathsf{adq})\ i$. We need to show that $h^\sharp(f^\sharp(d))\ \mathsf{adq}\ (e;g);i$. Computing the denotational semantics and using the fact that we may reassociate sequences **(8.2.4∗8)**, it suffices to show $((h^\sharp \circ f)^\sharp(d))\ \mathsf{adq}\ (e;(\lambda v.\ g\ v;i))$. By the assumption that $d \lhd_{\mathsf{UF}A} e$, it

suffices to show that for all $a \lhd_A v$, we have that $(h^\sharp(f\ a))\ \mathsf{adq}\ (g\ v; i)$, which follows directly from the assumptions that $f \lhd_{\mathsf{U}(A\to X)} g$ and $h\ (\lhd_B \Rightarrow \mathsf{adq})\ i$.

2. If $X = B \to Y$, suppose that $b \lhd_B v$. We need to show that $(f^\sharp(d))\ b \lhd_{\mathsf{U}Y} (e; g)\ v$. Unraveling the denotational semantics and the computational semantics using **(8.2.4∗12)**, it suffices to show $(\lambda d.\ f\ d\ b)^\sharp\ d \lhd_{\mathsf{U}Y} (e; \lambda d.\ g\ d\ v)$, which follows from the inductive hypothesis and the assumption that $f \lhd_{\mathsf{U}(A\to(B\to Y))} g$.

☐ **(8.4.2∗8)** If $d \lhd_X e$, then $c \boxplus d \lhd_X \mathsf{step}^c(e)$.

■ **(8.4.2∗9)** Since $\mathsf{step}^c(e) \mapsto c, e$, the result holds by **(8.4.2∗2)**.

☐ **(8.4.2∗10)** For every closed term $e : \Gamma \vdash A$, the approximation $\Gamma \vdash [\![e]\!] \lhd_A e$ holds.

■ **(8.4.2∗11)** By **(8.4.1∗5)**, **(8.4.2∗4)**, **(8.4.2∗6)** and **(8.4.2∗8)**.

» **(8.4.2∗12)** Given $e : \mathsf{UF1}$, we have that $[\![e]\!] = \mathsf{profile}(e)$.

(8.4.2∗13) In the functional phase both the denotational and computational semantics of e are simply partial computations of type $\mathsf{L1}$, so one may view **(8.4.2∗12)** as a cost-sensitive (and internal) version of Ploktin's original adequacy theorem for **PCF**.

8.5. INTERNAL AND EXTERNAL ADEQUACY

(8.5∗1) In Section 8.4 we proved an *internal* adequacy theorem about $\mathbf{PCF}_{\mathsf{cost}}$, a language defined internally to the synthetic domain theory topos. What does that say about "ordinary" $\mathbf{PCF}_{\mathsf{cost}}$, *i.e.* the inductive family specified by **(4.1.3∗9)** in **Set**.

(8.5∗2) Ideally, we would like an external version of **(8.4.2∗12)** about $\mathbf{PCF}_{\mathsf{cost}}$ taken as a language in **Set**. In the case of a sheaf model of synthetic domain theory, we may attempt to relate internal and external adequacy as follows, assuming that the coverage on the base domain-theoretic category $\mathscr{C}$ is subcanonical, *i.e.* admits a fully faithful embedding $\mathscr{C} \hookrightarrow \mathrm{Sh}(\mathscr{C})$ (which is satisfied by the model construction of Section 7.4). It is a routine exercise in Gödel numbering to see that internally to the synthetic domain theory topos, every $\mathbf{PCF}_{\mathsf{cost}}$ term can be assigned a unique number $n : \mathbb{N}$. Since every category of sheaves on $\mathscr{C}$ is reflective in $\widehat{\mathscr{C}}$ **(3.5.1∗7)** , the natural numbers object in $\mathrm{Sh}(\mathscr{C})$ can be computed by applying the sheafification functor $a : \widehat{\mathscr{C}} \to \mathrm{Sh}(\mathscr{C})$ to the natural numbers object $\mathbb{N}_{\widehat{\mathscr{C}}} = \Delta(\mathbb{N}_{\mathbf{Set}})$ in $\widehat{\mathscr{C}}$, where $\Delta : \mathbf{Set} \to \widehat{\mathscr{C}}$ is the constant presheaf functor. Therefore we have the following series of bijective correspondences.

$$\frac{\dfrac{\dfrac{\dfrac{1_{\mathrm{Sh}(\mathscr{C})} \longrightarrow \mathbb{N}_{\mathrm{Sh}(\mathscr{C})} \text{ in } \mathrm{Sh}(\mathscr{C})}{1_{\mathrm{Sh}(\mathscr{C})} \longrightarrow a(\Delta\mathbb{N}_{\mathbf{Set}}) \text{ in } \mathrm{Sh}(\mathscr{C})}}{a(\Delta 1_{\mathbf{Set}}) \longrightarrow a(\Delta\mathbb{N}_{\mathbf{Set}}) \text{ in } \mathrm{Sh}(\mathscr{C})}}{\Delta 1_{\mathbf{Set}} \longrightarrow \Delta\mathbb{N}_{\mathbf{Set}} \text{ in } \widehat{\mathscr{C}}}}{1_{\mathbf{Set}} \longrightarrow \mathbb{N}_{\mathbf{Set}} \text{ in } \mathbf{Set}}$$

In the above we used the fact that the associated sheaf functor $a : \widehat{\mathscr{C}} \to \mathrm{Sh}(\mathscr{C})$ is lex (preserves finite limits) and so $1_{\mathrm{Sh}}(\mathscr{C}) \cong a(1_{\widehat{\mathscr{C}}}) = a(\Delta 1_{\mathbf{Set}})$ and the fact that it is also fully faithful since the coverage is subcanonical. Thus we see that there is a bijective correspondence between global elements of internal terms $e : 1 \to \mathsf{tm}(\Gamma, A)$ and external terms $\ulcorner e \urcorner : 1 \to \mathbb{N}_{\mathbf{Set}}$. Observe that we may define an *external* profiling cost semantics $\ulcorner e \urcorner \Downarrow^c$. Given an external natural number c, *external adequacy* is the property that $[\![e]\!]$ denotes the numeral $\bar{c}$ if and only if $\ulcorner e \urcorner \Downarrow^c$.

(8.5∗3) While we conjecture that the backwards direction would follow from a simple external induction on the derivation of $\ulcorner e \urcorner \Downarrow^c$, the forwards direction requires further investigation of the synthetic ω-joins. By internal adequacy **(8.4.2∗12)**, it suffices to show that $[\![e]\!] = \mathsf{profile}(e)$ implies $\ulcorner e \urcorner \Downarrow^c$. Because the termination support of $\mathsf{profile}(e)$ is defined as a synthetic ω-join of Σ-propositions, we must show that when this internal join $\bigvee \phi_i$ holds, we have $\ulcorner e \urcorner \Downarrow^c$ as well, where $\bar{c}$ is the cost denoted by $\mathsf{profile}(e)$. Since synthetic ω-joins are unlikely to be preserved by $\Sigma \hookrightarrow \Omega$, it is unclear how to give an external characterization of the internal join $\bigvee \phi_i$ (for instance, it is probably *not* the case that $\bigvee \phi_i = \top$ implies there is a global element $i : 1 \to \omega$ such that $\phi_i = \top$). In the absence of such a characterization it would be difficult to relate the internal operational cost semantics $\mathsf{profile}(e)$ and its external counterpart $\ulcorner e \urcorner \Downarrow^c$.

(8.5∗4) Simpson introduced several important ideas and techniques for developing computational adequacy proofs internal to a topos. In particular a general property of the internal logic of topoi called *1-consistency*[1] is proved to be both necessary and sufficient to relate the internal adequacy property to an external adequacy property in the sense of **(0.5∗3)**.

(8.5∗5) However Simpson [114] assumes that $\mathbb{N}$ is well-complete, which closes the dominance Σ under countable joins of decidable families in the ambient logic. We do not rely on this axiom in the constructions of this chapter, but under this axiom one may define the internal computational dynamics of $\mathbf{PCF}_{\mathsf{cost}}$ **(8.2.4∗2)** by means of existentially quantified statements of the form $\exists[n : \mathbb{N}]\ \phi(n)$ where ϕ is a primitive recursive predicate, thence 1-consistency can be used to externalize the adequacy property.

(8.5∗6) Consequently, one way out of the conundrum of **(8.5∗3)** would be to alter the model construction of Section 7.4 so that natural numbers of the SDT topos is a predomain, *i.e.* equip our domain-theoretic site with the countable extensive coverage so that the natural numbers object, which is the countable coproduct $\amalg_{n \in \mathbb{N}} 1$, is preserved by the Yoneda embedding. Under this coverage the lift functor need not preserve ω-filtered colimits (essentially because one can no longer exploit the commutation of *finite* limits and filtered colimits), which means one loses the characterization of the initial lift algebra as an *inductive* fixed-point object **(7.2.3∗13)**. Nonetheless since the property that the initial lift algebra is computed as the colimit of the standard chain is only used to show that every representable presheaf is a well-complete object, one may try to show that the lift functor preserves κ-filtered colimits for some cardinal $\kappa > \omega$ and replay the same argument by exhibiting the initial lift algebra as a κ-filtered colimit of a "longer" chain obtained by transfinitely iterating the lift functor. Thus it appears there are no serious obstructions in updating and validating the sheaf model construction of Sterling and Harper [122] against this coverage, but we leave this to future work.

(8.5∗7) In an effort to reduce the reliance on the ambient nno of the SDT topos, Simpson in a follow up paper [113] to Simpson [114] develops an alternative way to relate internal and external adequacy by means of the notion of *computational 1-consistency*, a logical principle analogous to

[1]A topos $\mathcal{E}$ is *1-consistent* when a closed formula $\exists[n : \mathbb{N}]\ \phi(n)$ of the form described above holding in the internal logic of $\mathscr{E}$ implies that it holds externally.

1-consistency for the nno of predomains $\mathbb{N}_{\mathsf{P}}$. However, since computational 1-consistency is still a statement intrinsically about natural numbers, it is unclear how this can be applied to our setting, where the central objects such as the initial lift algebra ω and synthetic ω-chains do not have have natural external counterparts.

CHAPTER 9

Conclusion

(9∗1) In the first half of this dissertation, we saw that a number of known problems with incorporating cost into type theory may be resolved by imposing a (Kripke) world view consisting of a cost-sensitive phase lying over a purely functional phase, which is mathematically represented by the category of presheaves over the interval $\mathbb{I}$. This semantic picture gives rise to powerful restriction and sealing modalities in the internal language of $\widehat{\mathbb{I}}$ that one may use to organize cost structure by exploiting the configuration of the cost-sensitive and functional phases; in particular, by recording the cost of programs via a sealed monoid, one may simultaneously speak about a cost-sensitive function and restrict to its purely functional/mathematical aspect by means of the restriction modality as necessary. Combining this internal modal type theory of $\widehat{\mathbb{I}}$ with the idea of cost as an abstract effect, we obtain a type theory dubbed **calf** suitable for general cost-sensitive specification, programming, and verification. We selected some common textbook examples of algorithm analyses as case studies and showed that the usual pen-and-paper techniques carry over to **calf**. What is gained is 1) a higher level of rigor, 2) a means for compositional cost bounds, 3) the ability to mechanize the results in a computerized proof assistant.

(9∗2) In the second half we sought to relate denotational/equational reasoning in **calf** to operational cost semantics. The method is an internalization of the classic program of denotational (cost) semantics in which one specifies a programming language and its operational and denotational cost semantics as functions in the type theory. In order to study programming languages with general recursion, we axiomatize and work inside a cost-sensitive synthetic domain theory and proved a cost-sensitive computational adequacy result that restricts to the classic theorem of Plotkin's in the purely functional phase. We justified these internal constructions by showing that the relative sheaf model of synthetic domain theory of Sterling and Harper [122] is a model of cost-sensitive synthetic domain theory.

(9∗3) Now seems appropriate for some higher level remarks about the method and results of this dissertation. First is the importance of functional semantics. In the early years of my PhD I struggled to pin computational cost as a *property* of the execution behavior of raw syntax, which resulted in a type theory [88] that was intractable to use in practice. The remedy is to work not over raw syntax but over *equivalence classes of terms* or *functions* only. Rather than assigning properties to computations based on some accidental property (such as which raw term is chosen as the representative of the function), cost should be thought of as *structures over functions*, a lesson from workers in the metatheory of type theory.

(9∗4) The internal modal type theory resulting from the *Artin gluing* of topoi (of which $\widehat{\mathbb{I}}$ is a simple instance) was first applied to the theory of programming languages by Sterling and Harper in the context of program modules and data abstraction and has proved to be a versatile weapon against classic PL problems including cost analysis [90, 46], metatheory of (cubical, multimodal) dependent type theories [118, 116, 41], controlled unfolding of definitions in proof assistants [44], and information flow [122].

(9∗5) Lastly, the results of this dissertation (especially those of Chapters 7 and 8) contribute towards the growing evidence that despite their waning prevalence in the previous decades, denotational methods are increasingly germane in contemporary PL research. Although operational methods have been developed to spectacular heights and have dominated the general PL landscape in recent years, denotational semantics still has an important role as a space for developing general, reusable mathematical principles that may be combined to attack challenging problems. Following the renewed interests of the community in denotational methods, this dissertation can be thought of as a small step onto the path left by the semanticists of the 90s.

9.1. RELATED WORK

(9.1∗1) In this section I briefly survey other approaches to cost-sensitive verification and semantics, focusing on the lines of work most closely resembling this thesis either in application or technique.

9.1.1. Program logics.

(9.1.1∗1) The approach perhaps most closely related in terms of the target use of the framework has been developed in a line of work based on separation logic [100]. First noticed in Atkey [7] to have applications to resource analysis, separation logic has since been used as the theoretical basis of verification frameworks in a multitude of works expanding the boundaries of formalized algorithm analysis, including formalized proof of the correctness and tight cost bounds for the *union-find* data structure [22], analysis of heap space usage under garbage collection [79], and a formalization of the asymptotic notation [47].

(9.1.1∗2) From a technical point of view program logics are evidently different from type theories since there is a categorical separation between the programming language and the logical language in the former that is not present in the latter. Moreover, since the primary use of (concurrent) separation logic is the verification of (concurrent) imperative programs, works of this ilk tend towards the analysis of (concurrent) imperative algorithms, which I have not explored in this thesis.

Aside from requiring a much more heavy-handed model construction, directly verifying imperative programs in separation logic appears to be working at the incorrect abstraction level. In my view, it is more sensible to verify programs that are mathematically natural (*i.e.* functional) *first* and progressively lower the abstraction level by means of cost-sensitive adequacy theorems in the sense of Chapter 8 between *different* programming languages. Admittedly this is, however, pure speculation inspired by the results obtained in this thesis.

Despite the general technical differences between program logics and type theories, the works cited in **(9.1.1∗1)** are still amongst the most closely related works to this thesis since in addition to cost analysis they also handle general program verification.

9.1.2. Denotational cost semantics.

(9.1.2∗1) Denotational cost semantics of general recursion has been studied since at least the work of Plotkin and Power, who prove a general computational adequacy property for not just cost-sensitive **PCF** but (call-by-value) **PCF** equipped with any algebraic effect [95]. More recently Kavvos et al. present a formalization of the method of recurrence relations for cost analysis. Part of the semantic justifications of *op. cit.* consists of a denotational cost semantics for **PCF** in *sized domains*, structures that possess both the usual domain-theoretic information order and a *cost preorder* that interacts with the information order in a nontrivial way.

The biggest difference between the material in the second half of this thesis compared to these prior works is that in the former the denotational semantics is developed inside a type theory, whereas the latter take place in concrete categories of domains (or in order-enriched categories in the case of Plotkin and Power [95]). Thus the denotational semantics in the present work can be directly reasoned with in the same framework that one uses to write programs, which in my view represents an improvement over the status quo and in fact is a state of affairs that has been conjectured in Plotkin and Power [95, p. 2]. Moreover, by formulating cost structure in terms of a phase distinction, both the denotational cost semantics and the cost-sensitive computational adequacy property in Chapter 8 immediately restrict to an ordinary, functional semantics and the corresponding adequacy property in the functional phase, which I believe has not been possible until now.

9.1.3. Computational adequacy in synthetic domain theory.

(9.1.3∗1) The developments of Chapters 7 and 8 on synthetic domain theory and internal computational adequacy are heavily inspired by Simpson [114] and Simpson [112], from which many important definitions and mathematical techniques were borrowed. To adapt classic denotational semantics definitions into the synthetic setting, Simpson [112] has chosen to emphasize the role of the computational natural numbers (the natural numbers object in the category of predomains) in developing the syntax and operational semantics of the programming language. By contrast, in this thesis I have instead advocated for the use of the synthetic ωcpo structure of the initial lift algebra in defining the operational semantics, which is a natural choice because every model of synthetic domain theory must possess such an initial lift algebra (as shown in Section 7.3.6.3). I leave the relationship between these two approaches and the extension of the computational adequacy to recursive types to future work.

(9.1.3∗2) Another recent work I relied heavily on is Sterling and Harper [122], who studied the denotational semantics and computational adequacy of information flow security in synthetic domain theory. Indeed, thinking about the functional phase $\circ$ and cost-sensitive phase $\bullet$ of **calf** as a two-element security poset $\mathbb{I} = \{\circ \sqsubseteq \bullet\}$, the model construction of Section 7.4 is just an instantiation of the relative sheaf model of synthetic domain theory of *op. cit.* at $\mathbb{I}$. However, because in this thesis I have emphasized the intrinsic/synthetic order relation of predomains, some additional properties of the model had to be verified (such as Phoa's principle).

The approach to computational adequacy is also substantially different. In particular, while studying the computational adequacy proof of Sterling and Harper [122] Sterling and I discovered a nontrivial mistake in the construction of the formal approximation relation for free algebras [115]. The proof of computational adequacy in this thesis essentially originated from our effort to rescue the erroneous proof in Sterling and Harper [122]. Although it seems promising to adapt the results in Chapter 8 to the original setting of information flow security, it remains a challenge to design a programming language that itself incorporates phase propositions (as opposed to the case of $\mathbf{PCF}_{\mathsf{cost}}$ in which the phase proposition only appears in the denotational semantics).

9.1.4. Denotational semantics in synthetic guarded domain theory.

(9.1.4∗1) In addition to synthetic domain theory, synthetic *guarded* domain theory [14] has also been used to develop the denotational semantics of general recursion [93] and recursive types [84] inside *guarded* type theory. With mathematical roots in metric domain theory [4, 19], synthetic guarded domain theory can also be viewed as an abstraction of the well-known technique of step-indexing in constructing logical relations for recursive types [6] and high-order store [15].

In comparison to synthetic domain theory, synthetic guarded domain theory integrates better into type theory because *every* type can be regarded as a *guarded* predomain. Thus every guarded endomap whatsoever possesses a guarded fixed point, and this fact extends smoothly to universes, giving rise to a unified account of recursive functions and recursive types [131]. Combined with the fact that guarded predomains are closed under the usual type-theoretic constructions, synthetic guarded domain theory provides a robust mechanism for solving notoriously difficult domain equations. For instance, Sterling, Gratzer, and Birkedal [119] use synthetic guarded domain theory (along with ideas from realizability) to give a completely denotational account of the combination of parametric polymorphism and higher-order store, a result that has been difficult to obtain in classic domain theory.

(9.1.4∗2) However the great flexibility of synthetic guarded domain theory is not without drawbacks. The primary mathematical "defect" is that solutions to guarded fixed points are not fixed points on the nose; in particular the fixed-point equation only holds up to an abstract "step" that reflects the semantics of synthetic guarded domain theory in the *topos of trees* (the category of presheaves over $\omega = \{0 \sqsubseteq 1 \sqsubseteq \ldots\}$). Consequently the equational theory of the *guarded* lift monad is too fine to use directly in many situations since it distinguishes two partial elements denoting the same value but that differ in the number of abstract steps used to compute the value. Although this problem may be overcome by quotienting the type of partial computations by weak bisimulation, the resulting theory becomes needlessly involved in comparison to the canonical theory of fixed points furnished by synthetic domain theory (at least for the purposes of studying *functional* programs).

9.1.5. Type systems for cost analysis.

(9.1.5∗1) Lastly we discuss some related work that is outside of the realm of general-purpose verification and focuses more on pure cost analysis. Danielsson [30] introduces an indexed-monadic type system for tracking the cost of functional data structures. Like **calf**, this work treats cost structure as a computational effect and has been formalized in the Agda proof assistant. A major difference is that cost bounds are explicitly recorded in the *type* of the cost effect, a design choice that has caused well-known problems with making the specifications of data too precise in dependent type theory [30, Section 12]. For instance, the pervasive use of cost annotations for functions impedes composition across different modules and it is unclear if pure functional reasoning about programs is possible, despite the fact that the framework is essentially a library in a full-spectrum dependent type theory. This is not to claim that a theory of cost-bound composition at the level of *modules* has been worked out in **calf**, but at least the separation of code and their cost bounds means that composition of code would not present a problem in **calf**.

(9.1.5∗2) There has also been a variety of works on the automated side of formalized cost analysis [24, 97, 56, 59]. I will just focus on *Resource-Aware ML (RaML)* [56], which appears to be the most well-developed automated framework for deriving concrete cost bounds on functional programs. In brief, RaML enriches the type system of *e.g.* **PCF**[1] by assigning a *potential* to every type, resulting in a type-theoretic version of the amortized analysis of Tarjan [126]. The RaML framework collects linear constraints from the type checking process that are then delegated to an off-the-shelf linear program solver. When the constraint set is feasible, RaML is able to derive a cost bound that is (multi-variate) polynomial in the size of the input.

(9.1.5∗3) Clearly automated systems such as RaML and general-purpose verification frameworks like **calf** serve different purposes, but there appear to be opportunities for using one of the approaches to overcome limitations of the other and vice versa.

To set the stage, one may embed RaML into **calf** by means of a version of the internal denotational cost semantics defined in Chapter 8. In particular, I conjecture that one may prove a soundness theorem in which every cost bound derived by RaML is also valid in the denotational cost semantics.

The use of such an embedding is twofold. In one direction, one may enrich the RaML type system with what amounts to an *oracle* in order to derive cost bounds even in the presence of challenging functions that defeat automation. More specifically, one may provide RaML with cost bounds derived in **calf**, which may be treated as black-box facts from the point of view of RaML.

In the other direction, observe that under certain resource metrics, a cost bound on a program derived by RaML can be viewed as a *proof of termination*. Therefore, it would be reasonable to use RaML as a mechanism for programming and reasoning about general recursion in type theory. For instance, although merge sort may be seen (after some effort) to be a total function in **calf**, RaML would automatically derive a cost bound (and therefore a termination proof) for merge sort, thereby providing a way for programmers to write code in a natural manner that nonetheless can be seen to satisfy certain safety[2] properties.

9.2. FUTURE WORK

(9.2∗1) There are a number of directions in which it would be reasonable to extend the work of this thesis. Perhaps the most obvious extension would be the cost-sensitive computational adequacy property of recursive types. Although I have not focused on the order-enriched aspects of the model construction in Section 7.4, the construction would also suffice to furnish a denotational semantics for recursive types. The goal would then be to adapt the methods of Simpson [112] to account for the presence of cost structure; here the diverging approaches to the syntax and operational semantics of the internal programming language discussed in **(9.1.3∗1)** would need to be reconciled.

[1]RaML also supports recursive types.
[2]Safety in the colloquial sense, since termination is of course a *liveness* property.

(9.2∗2) A central position of this thesis is that cost profiling should not interfere with the pure functional behavior of programs — that there should be a phase distinction between the cost-sensitive world and the functional world. But this position only represents a certain platonic ideal that may be violated in practice. For instance, if the computational resource under analysis is the amount of physical memory available to a computer, information from the cost-sensitive phase may leak into the functional phase since repeated memory allocation without deallocation would eventually exhaust the memory limit and cause the program to crash or signal an error state (also known as a memory leak). It seems promising to consider the notion of *termination declassification* as in Sterling and Harper [122] to mediate this leakage of information; translated into the setting of cost analysis, one may use declassification to model a function that diverges in the cost-sensitive phase (due to limits on the computational resource) but appears to terminate in the functional phase.

Bibliography

[1] New Orleans, LA, USA: Association for Computing Machinery, 2020. ISBN: 978-1-4503-7097-4.

[2] Samson Abramsky and Achim Jung. “Domain Theory”. In: *Handbook of Logic in Computer Science (Vol. 3): Semantic Structures.* USA: Oxford University Press, Inc., 1995, pp. 1–168. ISBN: 0-19-853762-X.

[3] S. F. Allen et al. “Innovations in computational type theory using Nuprl”. In: *Journal of Applied Logic* 4.4 (2006). Towards Computer Aided Mathematics, pp. 428–469. ISSN: 1570-8683.

[4] Pierre America and Jan J. M. M. Rutten. “Solving Reflexive Domain Equations in a Category of Complete Metric Spaces”. In: *Proceedings of the 3rd Workshop on Mathematical Foundations of Programming Language Semantics.* Berlin, Heidelberg: Springer-Verlag, 1987, pp. 254–288. ISBN: 3-540-19020-1.

[5] Carlo Angiuli et al. “Syntax and models of Cartesian cubical type theory”. In: *Mathematical Structures in Computer Science* 31.4 (2021), pp. 424–468. DOI: `10.1017/S0960129521000347`.

[6] Andrew W. Appel and David McAllester. “An Indexed Model of Recursive Types for Foundational Proof-carrying Code”. In: *ACM Transactions on Programming Languages and Systems* 23.5 (Sept. 2001), pp. 657–683. ISSN: 0164-0925. DOI: `10.1145/504709.504712`.

[7] Robert Atkey. “Amortised Resource Analysis with Separation Logic”. In: *Programming Languages and Systems.* Ed. by Andrew D. Gordon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 85–103. ISBN: 978-3-642-11957-6.

[8] Steve Awodey. *Category Theory.* 2nd. New York, NY, USA: Oxford University Press, Inc., 2010. ISBN: 978-0-19-923718-0.

[9] Steve Awodey, Nicola Gambino, and Sina Hazratpour. *Kripke-Joyal forcing for type theory and uniform fibrations.* Unpublished manuscript. 2021. arXiv: `2110.14576 [math.LO]`.

[10] Steve Awodey and Florian Rabe. “Kripke Semantics for Martin-Löf’s Extensional Type Theory”. In: *Log. Methods Comput. Sci.* 7 (2009). URL: `https://api.semanticscholar.org/CorpusID:7424663`.

[11] Jon Beck. “Distributive laws”. In: *Seminar on Triples and Categorical Homology Theory.* Ed. by B. Eckmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 1969, pp. 119–140. ISBN: 978-3-540-36091-9.

[12] Marc Bezem, Thierry Coquand, and Simon Huber. "A Model of Type Theory in Cubical Sets". In: *19th International Conference on Types for Proofs and Programs (TYPES 2013)*. Ed. by Ralph Matthes and Aleksy Schubert. Vol. 26. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 107–128. ISBN: 978-3-939897-72-9. DOI: `10.4230/LIPIcs.TYPES.2013.107`. URL: `http://drops.dagstuhl.de/opus/volltexte/2014/4628`.

[13] Lars Birkedal et al. "First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees". In: *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. 2011, pp. 55–64. DOI: `10.1109/LICS.2011.16`.

[14] Lars Birkedal et al. "First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees". In: *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 55–64. ISBN: 978-0-7695-4412-0. DOI: `10.1109/LICS.2011.16`. arXiv: `1208.3596 [cs.LO]`.

[15] Lars Birkedal et al. "Step-Indexed Kripke Models over Recursive Worlds". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 119–132. ISBN: 978-1-4503-0490-0. DOI: `10.1145/1926385.1926401`.

[16] Aleš Bizjak and Rasmus Ejlers Møgelberg. "Denotational semantics for guarded dependent type theory". In: *Mathematical Structures in Computer Science* 30.4 (2020), pp. 342–378. DOI: `10.1017/S0960129520000080`.

[17] Guy Blelloch and John Greiner. "Parallelism in Sequential Functional Languages". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. La Jolla, California, USA: Association for Computing Machinery, 1995, pp. 226–237. ISBN: 0-89791-719-7. DOI: `10.1145/224164.224210`.

[18] Ana Bove and Venanzio Capretta. "Modelling general recursion in type theory". In: *Mathematical Structures in Computer Science* 15.4 (2005), pp. 671–708. DOI: `10.1017/S0960129505004822`.

[19] Franck Breugel and Jeroen Warmerdam. *Solving Domain Equations in a Category of Compact Metric Spaces*. Tech. rep. NLD: CWI (Centre for Mathematics and Computer Science), 1994.

[20] Kevin Buzzard, Johan Commelin, and Patrick Massot. "Formalising Perfectoid Spaces". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 299–312. ISBN: 978-1-4503-7097-4. DOI: `10.1145/3372885.3373830`.

[21] John Cartmell. "Generalised Algebraic Theories and Contextual Categories". PhD thesis. Oxford University, Jan. 1978.

[22] Arthur Charguéraud and François Pottier. "Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits". en. In: *Journal of Automated Reasoning* 62.3 (Mar. 2019), pp. 331–365. ISSN: 0168-7433, 1573-0670. DOI: `10.1007/s10817-017-9431-7`. URL: `http://link.springer.com/10.1007/s10817-017-9431-7` (visited on 09/22/2024).

[23] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press, 2013. ISBN: 0-262-02665-1.

[24] Ezgi Çiçek et al. "Relational Cost Analysis". In: *POPL 2017: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* Paris, France: Association for Computing Machinery, 2017, pp. 316–329. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009858.

[25] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. "The Taming of the Rew: A Type Theory with Computational Assumptions". In: *Proceedings of the ACM on Programming Languages.* POPL 2021 (2021). URL: https://hal.archives-ouvertes.fr/hal-02901011.

[26] The Coq Development Team. *The Coq Proof Assistant Reference Manual.* 2016.

[27] Thierry Coquand. "Canonicity and normalization for dependent type theory". In: *Theoretical Computer Science* 777 (2019). In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I, pp. 184–191. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2019.01.015. arXiv: 1810.09367 [cs.PL].

[28] Pierre-Louis Curien. "Substitution Up to Isomorphism". In: *Fundam. Inf.* 19.1-2 (Sept. 1993), pp. 51–85. ISSN: 0169-2968. URL: http://dl.acm.org/citation.cfm?id=175469.175471.

[29] Pierre-Louis Curien, Richard Garner, and Martin Hofmann. "Revisiting the categorical interpretation of dependent type theory". In: *Theoretical Computer Science* 546 (2014). Models of Interaction: Essays in Honour of Glynn Winskel, pp. 99–119. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2014.03.003.

[30] Nils Anders Danielsson. "Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '08. San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 133–144. ISBN: 9781595936899. DOI: 10.1145/1328438.1328457. URL: https://doi.org/10.1145/1328438.1328457.

[31] Tom de Jong. *Domain Theory in Constructive and Predicative Univalent Foundations.* 2023. DOI: 10.48550/ARXIV.2301.12405.

[32] Peter Dybjer. "Internal type theory". In: *Types for Proofs and Programs: International Workshop, TYPES '95 Torino, Italy, June 5–8, 1995 Selected Papers.* Ed. by Stefano Berardi and Mario Coppo. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 120–134. ISBN: 978-3-540-70722-6.

[33] Uli Fahrenberg et al. "Languages of higher-dimensional automata". In: *Mathematical Structures in Computer Science* 31.5 (2021), pp. 575–613. DOI: 10.1017/S0960129521000293.

[34] Marcelo P. Fiore. "Axiomatic Domain Theory in Categories of Partial Maps". PhD thesis. University of Edinburgh, Nov. 1994. URL: https://era.ed.ac.uk/handle/1842/406.

[35] Marcelo P. Fiore. *Enrichment and Representation Theorems for Categories of Domains and Continuous Functions.* Unpublished manuscript. 1996. URL: http://www.cl.cam.ac.uk/~mpf23/papers/ADT/rep.ps.gz.

[36] Marcelo P. Fiore. "Lifting as a KZ-doctrine". In: *Category Theory and Computer Science.* Ed. by David Pitt, David E. Rydeheard, and Peter Johnstone. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 146–158. ISBN: 978-3-540-44661-3.

[37] Marcelo P. Fiore, Andrew M. Pitts, and S. C. Steenkamp. "Quotients, inductive types, and quotient inductive types". In: *Logical Methods in Computer Science* Volume 18, Issue 2 (June 2022). DOI: `10.46298/lmcs-18(2:15)2022`.

[38] Marcelo P. Fiore and Gordon D. Plotkin. "An Extension of Models of Axiomatic Domain Theory to Models of Synthetic Domain Theory". In: *Computer Science Logic, 10th International Workshop, CSL '96, Annual Conference of the EACSL, Utrecht, The Netherlands, September 21-27, 1996, Selected Papers*. Ed. by Dirk van Dalen and Marc Bezem. Vol. 1258. Lecture Notes in Computer Science. Springer, 1996, pp. 129–149. DOI: `10.1007/3-540-63172-0\_36`.

[39] Marcelo P. Fiore and Giuseppe Rosolini. "Two models of synthetic domain theory". In: *Journal of Pure and Applied Algebra* 116.1 (1997), pp. 151–162. ISSN: 0022-4049. DOI: `10.1016/S0022-4049(96)00164-8`.

[40] Georges Gonthier. "Formal Proof — The Four-Color Theorem". In: *Notices of the AMS* 55.11 (2008). URL: `https://www.ams.org/notices/200811/tx081101382p.pdf`.

[41] Daniel Gratzer. "Normalization for Multimodal Type Theory". In: *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2022. DOI: `10.1145/3531130.3532398`.

[42] Daniel Gratzer. "Syntax and semantics of modal type theory". English. PhD thesis. Oct. 2023.

[43] Daniel Gratzer and Jonathan Sterling. *Syntactic categories for dependent type theory: sketching and adequacy*. Unpublished manuscript. 2020. arXiv: `2012.10783 [cs.LO]`.

[44] Daniel Gratzer et al. *Controlling unfolding in type theory*. Oct. 2022. DOI: `10.48550/arXiv.2210.05420`. arXiv: `2210.05420 [cs.LO]`.

[45] Harrison Grodin and Robert Harper. "Amortized Analysis via Coinduction". In: *10th Conference on Algebra and Coalgebra in Computer Science (CALCO 2023)*. Ed. by Paolo Baldan and Valeria de Paiva. Vol. 270. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 23:1–23:6. ISBN: 978-3-95977-287-7. DOI: `10.4230/LIPIcs.CALCO.2023.23`. URL: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CALCO.2023.23`.

[46] Harrison Grodin et al. "Decalf: A Directed, Effectful Cost-Aware Logical Framework". In: *Proceedings of the ACM on Programming Languages* 8.POPL (Jan. 2024). DOI: `10.1145/3632852`.

[47] Armaël Guéneau, Arthur Charguéraud, and François Pottier. "A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification". en. In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Vol. 10801. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 533–560. ISBN: 978-3-319-89883-4 978-3-319-89884-1. DOI: `10.1007/978-3-319-89884-1_19`. URL: `http://link.springer.com/10.1007/978-3-319-89884-1_19` (visited on 09/22/2024).

[48] Jesse Michael Han and Floris van Doorn. "A Formal Proof of the Independence of the Continuum Hypothesis". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 353–366. ISBN: 978-1-4503-7097-4. DOI: `10.1145/3372885.3373826`.

[49] Martin A. T. Handley, Niki Vazou, and Graham Hutton. “Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019). DOI: `10.1145/3371092`.

[50] Robert Harper. *An Equational Logical Framework for Type Theories*. 2021. arXiv: `2106.01484 [math.LO]`.

[51] Robert Harper. ***PFPL*** *Supplement: Types and Parallelism*. 2018. URL: `https://www.cs.cmu.edu/~rwh/pfpl/supplements/par.pdf`.

[52] Robert Harper. *Practical Foundations for Programming Languages*. First. New York, NY, USA: Cambridge University Press, 2012.

[53] Robert Harper. *Practical Foundations for Programming Languages*. Second. New York, NY, USA: Cambridge University Press, 2016.

[54] Robert Harper, Furio Honsell, and Gordon D. Plotkin. “A Framework for Defining Logics”. In: *Journal of the ACM* 40.1 (Jan. 1993), pp. 143–184. ISSN: 0004-5411. DOI: `10.1145/138027.138060`.

[55] Sina Hazratpour. “A logical study of some 2-categorical aspects of topos theory”. PhD thesis. University of Birmingham, 2019. URL: `https://etheses.bham.ac.uk//id/eprint/9752/7/Hazratpour2019PhD.pdf`.

[56] Jan Hoffmann. “Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis”. PhD thesis. Ludwig-Maximilians-Universität München, 2011. URL: `https://www.cs.cmu.edu/~janh/assets/pdf/Hoffmann11.pdf`.

[57] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. “Resource Aware ML”. In: *Computer Aided Verification*. Ed. by P. Madhusudan and Sanjit A. Seshia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 781–786. ISBN: 978-3-642-31424-7.

[58] Martin Hofmann. “On the interpretation of type theory in locally cartesian closed categories”. In: *Computer Science Logic*. Ed. by Leszek Pacholski and Jerzy Tiuryn. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 427–441. ISBN: 978-3-540-49404-1.

[59] Martin Hofmann and Steffen Jost. “Static Prediction of Heap Space Usage for First-Order Functional Programs”. In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New Orleans, Louisiana, USA: Association for Computing Machinery, 2003, pp. 185–197. ISBN: 1-58113-628-5. DOI: `10.1145/604131.604148`.

[60] Martin Hofmann and Thomas Streicher. “Lifting Grothendieck Universes”. Unpublished note. 1997. URL: `https://www2.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf`.

[61] Simon Huber. “Canonicity for Cubical Type Theory”. In: *Journal of Automated Reasoning* (June 13, 2018). ISSN: 1573-0670. DOI: `10.1007/s10817-018-9469-1`.

[62] J. M. E. Hyland. “First steps in synthetic domain theory”. In: *Category Theory*. Ed. by Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 131–156. ISBN: 978-3-540-46435-8.

[63] Bart Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141. Amsterdam: North Holland, 1999.

[64] Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium: Volumes 1 and 2.* Oxford Logical Guides 43. Oxford Science Publications, 2002.

[65] Ralf Jung et al. “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”. In: *Journal of Functional Programming* 28 (2018), e20. DOI: `10.1017/S0956796818000151`.

[66] G. A. Kavvos et al. “Recurrence Extraction for Functional Programs through Call-by-Push-Value”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019). DOI: `10.1145/3371083`.

[67] G M Kelly. “BASIC CONCEPTS OF ENRICHED CATEGORY THEORY”. In: ().

[68] Nicolas Koh et al. “From C to interaction trees: specifying, verifying, and testing a networked server”. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs.* CPP 2019. Cascais, Portugal: Association for Computing Machinery, 2019, pp. 234–248. ISBN: 9781450362221. DOI: `10.1145/3293880.3294106`. URL: `https://doi.org/10.1145/3293880.3294106`.

[69] Saul A. Kripke. “Semantical Analysis of Intuitionistic Logic I”. In: *Formal Systems and Recursive Functions.* Ed. by J. N. Crossley and M. A. E. Dummett. Vol. 40. Studies in Logic and the Foundations of Mathematics. Elsevier, 1965, pp. 92–130. DOI: `10.1016/S0049-237X(08)71685-9`.

[70] Daniel K. Lee, Karl Crary, and Robert Harper. “Towards a Mechanized Metatheory of Standard ML”. In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* Nice, France: Association for Computing Machinery, 2007, pp. 173–184. ISBN: 1-59593-575-4. DOI: `10.1145/1190216.1190245`.

[71] Xavier Leroy et al. “CompCert – A Formally Verified Optimizing Compiler”. In: *ERTS 2016: Embedded Real Time Software and Systems.* SEE, 2016. URL: `http://xavierleroy.org/publi/erts2016_compcert.pdf`.

[72] Paul Blain Levy. “Call-By-Push-Value”. PhD thesis. 2001. URL: `https://www.cs.bham.ac.uk/~pbl/papers/thesisqmwphd.pdf`.

[73] Paul Blain Levy. *Call-by-Push-Value: A Functional/Imperative Synthesis.* Kluwer, Semantic Structures in Computation, 2, Jan. 1, 2003. ISBN: 1-4020-1730-8.

[74] Paul Blain Levy. “Call-by-push-value: Decomposing call-by-value and call-by-name”. In: *Higher-Order and Symbolic Computation* 19 (2006), pp. 377–414. DOI: `10.1007/s10990-006-0480-6`.

[75] Runming Li, Harrison Grodin, and Robert Harper. *A Verified Cost Analysis of Joinable Red-Black Trees.* 2023. arXiv: `2309.11056`.

[76] F. E. J. Linton. “Coequalizers in categories of algebras”. In: *Seminar on Triples and Categorical Homology Theory.* Ed. by B. Eckmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 1969, pp. 75–90. ISBN: 978-3-540-36091-9.

[77] John R. Longley and Alex K. Simpson. “A uniform approach to domain theory in realizability models”. In: *Mathematical Structures in Computer Science* 7.5 (1997), pp. 469–505. DOI: `10.1017/S0960129597002387`.

[78] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in geometry and logic: a first introduction to topos theory.* Universitext. New York: Springer, 1992. ISBN: 0-387-97710-4.

[79] Jean-Marie Madiot and François Pottier. *A Separation Logic for Heap Space under Garbage Collection.* Conditionally accepted to POPL '22. 2021. URL: `http://cambium.inria.fr/~fpottier/publis/madiot-pottier-diamonds-2021.pdf`.

[80] Michael Makkai and Giuseppe Rosolini. "Studying Repleteness in the Category of Cpos". In: *Electronic Notes in Theoretical Computer Science.* MFPS XIII, Mathematical Foundations of Progamming Semantics, Thirteenth Annual Conference 6 (Jan. 1997), pp. 249–254. ISSN: 1571-0661. DOI: `10.1016/S1571-0661(05)80157-4`. (Visited on 12/27/2023).

[81] Per Martin-Löf. "Constructive Mathematics and Computer Programming". In: *6th International Congress for Logic, Methodology and Philosophy of Science.* Published by North Holland, Amsterdam. 1982. Hanover, Aug. 1979, pp. 153–175.

[82] Cristina Matache, Sean Moss, and Sam Staton. "Recursion and Sequentiality in Categories of Sheaves". In: *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).* Ed. by Naoki Kobayashi. Vol. 195. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 25:1–25:22. ISBN: 978-3-95977-191-7. DOI: `10.4230/LIPIcs.FSCD.2021.25`. URL: `https://drops.dagstuhl.de/opus/volltexte/2021/14263`.

[83] Glen Mével, Jacques-Henri Jourdan, and François Pottier. "Time Credits and Time Receipts in Iris". In: *Programming Languages and Systems.* Ed. by Luís Caires. Cham: Springer International Publishing, 2019, pp. 3–29. ISBN: 978-3-030-17184-1.

[84] Rasmus Ejlers Møgelberg and Marco Paviotti. "Denotational Semantics of Recursive Types in Synthetic Guarded Domain Theory". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science.* New York, NY, USA: Association for Computing Machinery, 2016, pp. 317–326. ISBN: 978-1-4503-4391-6. DOI: `10.1145/2933575.2934516`.

[85] Eugenio Moggi. "Notions of computation and monads". In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 55–92. ISSN: 0890-5401. DOI: `10.1016/0890-5401(91)90052-4`.

[86] Leonardo de Moura et al. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings.* Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21401-6.

[87] Yue Niu and Robert Harper. "A Metalanguage for Cost-Aware Denotational Semantics". In: *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS).* 2023, pp. 1–14. DOI: `10.1109/LICS56636.2023.10175777`.

[88] Yue Niu and Robert Harper. *Cost-Aware Type Theory.* 2020. arXiv: `2011.03660 [cs.PL]`.

[89] Yue Niu, Jonathan Sterling, and Robert Harper. *Cost-sensitive computational adequacy of higher-order recursion in synthetic domain theory.* 40th Conference on Mathematical Foundations of Programming Semantics (MFPS XXXX). 2024. arXiv: `2404.00212 [cs.PL]`. URL: `https://arxiv.org/abs/2404.00212`.

[90] Yue Niu et al. "A Cost-Aware Logical Framework". In: *Proceedings of the ACM on Programming Languages* 6.POPL (Jan. 2022). DOI: `10.1145/3498670`. arXiv: `2107.04663 [cs.PL]`.

[91] Ulf Norell. "Dependently Typed Programming in Agda". In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation.* TLDI '09. Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 1–2. ISBN: 978-1-60558-420-1.

[92] Dito Pataraia. *A constructive proof of Tarski's fixed-point theorem for dcpo's.* 65th Peripatetic Seminar on Sheaves and Logic. Aarhus, Denmark, Nov. 1997.

[93] Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. "A Model of PCF in Guarded Type Theory". In: *Electronic Notes in Theoretical Computer Science* 319.Supplement C (2015). The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI), pp. 333–349. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2015.12.020`.

[94] Wesley Phoa. "Domain Theory in Realizability Toposes". PhD thesis. University of Edinburgh, July 1991.

[95] Gordon Plotkin and John Power. "Adequacy for Algebraic Effects". en. In: *Foundations of Software Science and Computation Structures.* Ed. by Gerhard Goos et al. Vol. 2030. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–24. ISBN: 978-3-540-41864-1 978-3-540-45315-4. DOI: `10.1007/3-540-45315-6_1`. URL: `http://link.springer.com/10.1007/3-540-45315-6_1` (visited on 09/22/2024).

[96] Gordon D. Plotkin. "LCF considered as a programming language". In: *Theoretical Computer Science* 5.3 (1977), pp. 223–255. ISSN: 0304-3975. DOI: `10.1016/0304-3975(77)90044-5`.

[97] Vineet Rajani et al. "A Unifying Type-Theory for Higher-Order (Amortized) Cost Analysis". In: *Proceedings of the ACM on Programming Languages* 5.POPL (Jan. 2021). DOI: `10.1145/3434308`.

[98] Bernhard Reus. "Program Verification in Synthetic Domain Theory". PhD thesis. München: Ludwig-Maximilians-Universität München, Nov. 1995.

[99] Bernhard Reus and Thomas Streicher. "General synthetic domain theory — a logical approach". In: *Mathematical Structures in Computer Science* 9.2 (1999), pp. 177–223. DOI: `10.1017/S096012959900273X`.

[100] J.C. Reynolds. "Separation logic: a logic for shared mutable data structures". en. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science.* Copenhagen, Denmark: IEEE Comput. Soc, 2002, pp. 55–74. ISBN: 978-0-7695-1483-3. DOI: `10.1109/LICS.2002.1029817`. URL: `http://ieeexplore.ieee.org/document/1029817/` (visited on 09/22/2024).

[101] John C. Reynolds. "Types, Abstraction, and Parametric Polymorphism". In: *Information Processing.* 1983.

[102] Egbert Rijke, Michael Shulman, and Bas Spitters. "Modalities in homotopy type theory". In: *Logical Methods in Computer Science* 16 (1 Jan. 2020). DOI: `10.23638/LMCS-16(1:2)2020`.

[103] Guiseppe Rosolini. "Continuity and effectiveness in topoi". PhD thesis. University of Oxford, 1986.

[104] Peter Scholze. “Liquid Tensor Experiment”. In: *Experimental Mathematics* 31.2 (2022), pp. 349–354. DOI: `10.1080/10586458.2021.1926016`.

[105] Dana Scott and C. Strachey. “Towards a Mathematical Semantics for Computer Languages”. In: *Proceedings of the Symposium on Computers and Automata* 21 (Jan. 1971).

[106] Dana S. Scott. “Data Types as Lattices”. In: *SIAM Journal on Computing* 5.3 (1976), pp. 522–587. DOI: `10.1137/0205037`.

[107] Dana S. Scott. *Outline of a Mathematical Theory of Computation.* Tech. rep. PRG02. Oxford University Computer Laboratory, Nov. 1970, p. 30.

[108] R. A. G. Seely. “Locally cartesian closed categories and type theory”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 95.1 (1984), pp. 33–48. DOI: `10.1017/S0305004100061284`.

[109] Michael Shulman. *Internalizing the External, or The Joys of Codiscreteness.* blog. 2011. URL: `https://golem.ph.utexas.edu/category/2011/11/internalizing_the_external_or.html`.

[110] Michael Shulman. *Localization as an Inductive Definition.* blog. Dec. 6, 2011. URL: `https://homotopytypetheory.org/2011/12/06/inductive-localization/`.

[111] Michael Shulman. *Reflective Subfibrations, Factorization Systems, and Stable Units.* blog. Dec. 6, 2011. URL: `https://golem.ph.utexas.edu/category/2011/12/reflective_subfibrations_facto.html`.

[112] Alex Simpson. “Computational adequacy for recursive types in models of intuitionistic set theory”. In: *Annals of Pure and Applied Logic* 130.1 (2004). Papers presented at the 2002 IEEE Symposium on Logic in Computer Science (LICS), pp. 207–275. ISSN: 0168-0072. DOI: `10.1016/j.apal.2003.12.005`.

[113] Alex Simpson. “Semantic Structure for Programming Languages with Effects”. Lecture notes. Luminy, France, 2014. URL: `http://homepages.inf.ed.ac.uk/als/luminy.pdf`.

[114] Alex K. Simpson. “Computational Adequacy in an Elementary Topos”. In: *Computer Science Logic.* Ed. by Georg Gottlob, Etienne Grandjean, and Katrin Seyr. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1999, pp. 323–342. ISBN: 978-3-540-48855-2. DOI: `10.1007/10703163_22`.

[115] Jonathan Sterling. *Erratum: adequacy of* Sheaf semantics of noninterference. July 17, 2023. URL: `http://www.jonmsterling.com/jms-005Z.xml`.

[116] Jonathan Sterling. “First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory”. Version 1.1, revised May 2022. PhD thesis. Carnegie Mellon University, 2021. DOI: `10.5281/zenodo.6990769`.

[117] Jonathan Sterling. *Tensorial structure of the lifting doctrine in constructive domain theory.* 2024. arXiv: `2312.17023 [math.CT]`. URL: `https://arxiv.org/abs/2312.17023`.

[118] Jonathan Sterling and Carlo Angiuli. “Normalization for Cubical Type Theory”. In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS).* Los Alamitos, CA, USA: IEEE Computer Society, July 2021, pp. 1–15. DOI: `10.1109/LICS52264.2021.9470719`. arXiv: `2101.11479 [cs.LO]`.

[119] Jonathan Sterling, Daniel Gratzer, and Lars Birkedal. "Denotational semantics of general store and polymorphism". Unpublished manuscript. July 2022. DOI: `10.48550/arXiv.2210.02169`.

[120] Jonathan Sterling and Robert Harper. "Guarded Computational Type Theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '18. Oxford, United Kingdom: ACM, 2018, pp. 879–888. ISBN: 978-1-4503-5583-4. DOI: `10.1145/3209108.3209153`. URL: `http://doi.acm.org/10.1145/3209108.3209153`.

[121] Jonathan Sterling and Robert Harper. "Logical Relations as Types: Proof-Relevant Parametricity for Program Modules". In: *Journal of the ACM* 68.6 (Oct. 2021). ISSN: 0004-5411. DOI: `10.1145/3474834`. arXiv: `2010.08599 [cs.PL]`.

[122] Jonathan Sterling and Robert Harper. "Sheaf semantics of termination-insensitive noninterference". In: *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. Ed. by Amy P. Felty. Vol. 228. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Aug. 2022, 5:1–5:19. ISBN: 978-3-95977-233-4. DOI: `10.4230/LIPIcs.FSCD.2022.5`. arXiv: `2204.09421 [cs.PL]`.

[123] Thomas Streicher. *Domain-Theoretic Foundations of Functional Programming*. Dec. 2006. ISBN: 978-981-270-142-8. DOI: `10.1142/6284`.

[124] Aaron Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, 2016. ISBN: 978-1-970001-27-3.

[125] W. W. Tait. "Intensional Interpretations of Functionals of Finite Type I". In: *The Journal of Symbolic Logic* 32.2 (1967), pp. 198–212. ISSN: 00224812. URL: `http://www.jstor.org/stable/2271658`.

[126] R. Tarjan. "Amortized Computational Complexity". In: *Siam Journal on Algebraic and Discrete Methods* 6 (1985), pp. 306–318.

[127] Paul Taylor. "The fixed point property in synthetic domain theory". In: *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. 1991, pp. 152–160. DOI: `10.1109/LICS.1991.151640`.

[128] Sebastian Andreas Ullrich. "Simple Verification of Rust Programs via Functional Purification". MA thesis. IPD Snelting, Dec. 2016.

[129] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `https://homotopytypetheory.org/book`, 2013.

[130] Jaap van Oosten and Alex K. Simpson. "Axioms and (counter)examples in synthetic domain theory". In: *Annals of Pure and Applied Logic* 104.1 (2000), pp. 233–278. ISSN: 0168-0072. DOI: `10.1016/S0168-0072(00)00014-2`.

[131] Andrea Vezzosi. "Guarded Recursive Types in Type Theory". 63. PhD thesis. Institutionen för data- och informationsteknik, Datavetenskap (Chalmers), Chalmers tekniska högskola, 2015.

[132] Peng Wang, Di Wang, and Adam Chlipala. "TiML: A Functional Language for Practical Complexity Analysis with Invariants". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017). DOI: `10.1145/3133903`.

[133] Shao Zhong and Ford Bryan. *Advanced Development of Certified OS Kernels*. 2010. URL: `https://flint.cs.yale.edu/certikos/publications/ctos.pdf`.