Intensional Investigations

Denis R. Dancanet

CMU-CS-98-135

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

Thesis Committee:

Stephen Brookes, Chair
Guy Blelloch
Dana Scott
Gérard Berry, Ecole des Mines de Paris

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

This research was sponsored in part by the Office of Naval Research under Grant No. N00014-93-1-0750.

Keywords: semantics of programming languages, parallel programming languages, intensional semantics, intensional expressiveness, circuit semantics, circuit complexity, type inference, refinement types, concrete data structures, sequential algorithms, categorical combinators.

Abstract

This thesis is about the theory and practice of intensional semantics. Traditional denotational models of programming languages are usually extensional in that they concern themselves only with input/output properties of programs. The meaning of a program is typically taken to be a function from input to output containing no information about the way that function computes its result. In an intensional denotational semantics, the meaning of a program is an object embodying aspects of the computation strategy. The structure of the object varies, depending on the language one models and the intended usage. For instance, previous intensional semantics have been developed using functions on richer domains, pairs of a function and a computation strategy, and sequential algorithms, and they were used to reason about efficiency, complexity, order of evaluation, degrees of parallelism, efficiency-improving program transformations, and so on.

In the first part of this thesis, we develop an intensional semantics based on abstract circuits. A program is mapped to a circuit, whose dimensions tell us how much parallel work and time is required to execute the program. We relate the circuit dimensions to various execution strategies, and to more traditional models of parallel execution, such as the PRAM. Our main application for circuit semantics is the establishment of relative intensional expressiveness results. Extensional expressiveness is concerned with whether a construct enables us to compute new functions. Since most programming languages are Turing-complete this is usually not very interesting. On the other hand, intensional expressiveness is concerned with whether a construct enables us to write more efficient programs. Utilizing a somewhat surprising connection with the field of circuit complexity, we study the relative intensional expressive power of various deterministic and nondeterministic parallel extensions of PCF.

Although most of our results have to do with parallel programming languages, we also study relative intensional expressiveness in a sequential setting. Using techniques different from circuit semantics, we compare Colson's primitive recursive algorithms to Berry and Curien's sequential algorithms, in the area of efficient expressibility of a function that computes the minimum of two lazy natural numbers.

In the second part of this thesis, we establish the practical utility of intensional semantics, by taking an existing semantics, that of sequential algorithms on concrete data structures, and using it to develop a refinement type inference system. The system features recursive types, subtyping, intersection types, polymorphism, and overloading. The types are the concrete data structures, and the terms are expressions in a lazy, higher-order, polymorphic, functional language, which are compiled to categorical combinators represented by sequential algorithms. A type may be refined by several subtypes (for instance, *bool* can be refined by *true* and *false*). The type always differs from its refinements at a finite number of points. If a term has a regular type, then the system enters into an interrogative abstract interpretation session with it, seeking to evaluate it only at those points relevant from the point of view of refinement type inference. Sequential algorithms provide very precise information about the dependence of pieces of output on pieces of input, and we can use this intensional information to generate a refinement type. We prove soundness of both the type inference and refinement type inference, and we show several examples from our prototype implementation.

To my father and to the memory of my grandfather

Contents

Acknowledgements

First and foremost, I thank my advisor, Stephen Brookes, for being my teacher and mentor. He put up with my sketchy ideas and flippant writing style, and above all, he was patient. His demand for precision provided a great example. Naturally, any errors remaining in this document are his fault. (Just kidding, Steve.)

I thank Guy Blelloch, Dana Scott, and Gérard Berry for their comments and discussions about my work, and for serving on the thesis committee. I thank Peter Lee, Frank Pfenning, and John Reynolds for showing an interest in my work. On several occasions, Peter has come up with references that proved very relevant. Frank kindly explained to me his work on refinement type inference. In addition, I thank the rest of the members of the POP group for making CMU a stimulating place to do research in programming languages.

I thank Lokendra Shastri, my advisor at the University of Pennsylvania, for introducing me to research in computer science. I owe a great debt of gratitude to Val Breazu-Tannen, for his teaching, his advice, and his friendship.

It is with great pleasure that I acknowledge the help received from Sharon Burks, Catherine Copetas, Karen Olack, and the rest of the School of Computer Science support staff. They have created a wonderful environment.

A good deal of the fun in being a graduate student came from having a great group of officemates: Jefferey Shufelt, my closest friend during these years, with whom I generated an enormous number of ideas that will one day change the world, See-Kiong Ng, Jiři Sgall, Conrad Poelman, Peter Jansen. I thank you. Thanks also to Sue Older, Jürgen Dingel, Yuan Hsieh, and the other friends in the department.

Finally, I thank Nury Garcia for alternatively distracting me and encouraging me to work, and my father for his help, both spiritual and financial, without which this thesis would not have been possible. He remains the most interesting person I know, and this work is partially dedicated to him.

Pittsburgh, PA
August 1, 1997

Chapter 1

Introduction

This thesis explores theoretical and practical issues in the semantics of programming languages. On the theoretical side, we compare various sequential and parallel programming languages, with the aim of establishing when one allows us to write more efficient programs than another. We call this pursuit *relative intensional expressiveness*, and the main tool we use to achieve results is *intensional semantics*. Generally speaking, an intensional semantics is any semantics mapping a program into an object which provides some insight into the way the program computes its result, that is, the *computation strategy* of the program.

Since an intensional semantics provides information about the computation strategy of a program, the question naturally arises of how to take advantage of such information for the purpose of program analysis. On the practical side of this thesis, we show how to do this by designing a refinement type inference system using sequential algorithms on concrete data structures.

Before going any further, we shall describe in more detail what we mean by intensional semantics, relative intensional expressiveness, and refinement types.

1.1 Intensional semantics

The word *intension* is a loaded one in computer science in general, and even in the area of programming languages in particular. First, we give a brief description of the various usages of the word, pointing out the intended one in this work, followed by a discussion of intensional semantics proper, and a simple example.

1.1.1 The extension of intension

The word originated in philosophy: *intension* is the set of all attributes thought of as essential to the meaning of a term, as opposed to *extension*, which is the set of objects to which a term applies. It is used in logic: *intensional logic* is the branch of logic concerned with assertions whose meaning is dependent on an implicit context. The logic usage led to one of the meanings in the programming languages community: an *intensional programming language* is one with context dependent constructs (for instance, a notion of execution time step). The first such language was Lucid, occurring in both sequential [31] and parallel [4] flavors.

Another meaning of intension/extension is the opposition between the function-as-a-program / function-as-a-graph views. It is used this way in recursion theory and proof theory, and this is the intended meaning here. An *intensional programming language* is one with constructs which make explicit intensional properties, such as order of evaluation, degree of parallelism, etc. An example of such a language is Berry and Curien's CDS0 [7], a programming language of sequential algorithms on concrete data structures.

1.1.2 What is intensional semantics?

Traditionally, denotational semantics has mainly been used to reason about extensional properties of programs. The meaning of a program is typically taken to be a function from input to output conveying no information about the way that function computes its result. For instance, two sorting programs, such as bubblesort and mergesort, are mapped by an extensional semantics to the same input/output function, the function that sorts its input. However, the two programs are very different in terms of efficiency. This is an intensional feature. In an intensional denotational semantics, the meaning of a program is an object embodying aspects of the program's *computation strategy*, *i.e.,* the way the program computes its result, and thus by choosing an appropriate intensional model, bubblesort and mergesort can be differentiated. Ideally, one would like to be able to use an intensional semantics to establish relative efficiency results, *e.g.,* mergesort is "better" than bubblesort in average or worst case.

There are many ways of constructing intensional denotational models. We outline just a few possibilities:

- We could take the meaning of a program to be a function on a richer domain (*e.g.,* [11, 19]) whose structure permits us to deduce information about computation strategy. This is usually achieved by introducing partially defined elements in the model; by knowing what our program does on partial inputs, we can get an idea of the evaluation strategy.

- We could take the meaning to be a pair consisting of a function and an object conveying intensional information; this object could represent the cost of evaluating the function, or could be a function from inputs to costs (*e.g.,* [44, 77]).

- We could dispense with functions as meanings altogether, and use *algorithms* instead (*e.g.,* the Berry-Curien intensional model for sequential languages using sequential algorithms on concrete data structures [6]).

An important point to note is that intensionality is relative. A semantics can be more intensional than another one. For each extensional semantics there is a hierarchy of intensional semantics that add more and more information. Our choice of an intensional semantics should be dependent on what program properties we wish to reason about, *i.e.,* we should be able to pick the relevant amount of detail for the problem at hand.

When one is not interested in the intensional aspects of program behavior, the intensional model should agree with the extensional one. In other words, one should be able to throw away the extra detail in an intensional model (*e.g.,* the computation strategy) and have it *collapse* onto an extensional model. If our intensional models are such "conservative extensions" of the extensional one, then we could reason about both intensional and extensional aspects at the same time.

We will be referring to an intensional denotational semantics simply as an intensional semantics, although in general, an intensional semantics is any semantics which enables one to reason about intensional features. In particular, operational semantics has also been used to reason about intensional issues [75, 9, 41]. We are particularly interested in denotational models because they are defined *compositionally* and permit algebraic reasoning (to show that two programs have the same meaning, we need only show that they have the same denotation), and they enable the use of well-known techniques for reasoning about programs, such as fixed-point induction [80].

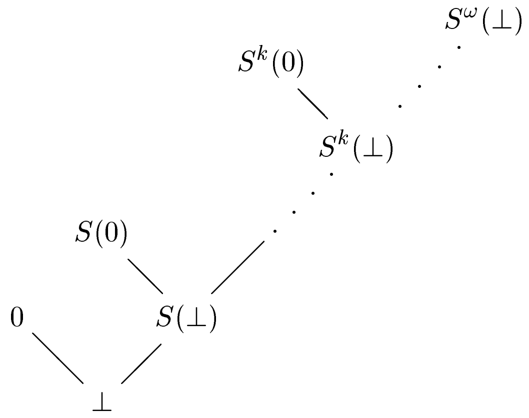

Figure 1.1: The lazy natural numbers

1.1.3 An example: Primitive recursion and the lazy natural numbers

For a simple example of an intensional semantics, consider the semantics of primitive recursive ($\mathcal{PR}$) algorithms. $\mathcal{PR}$ algorithms are just syntax for expressing $\mathcal{PR}$ functions [57]. The syntax is in the form of a rewrite system obeying certain syntactic restrictions corresponding to the format of primitive recursive function definitions (see Colson [19, 20] for a formal definition). The $\mathcal{PR}$ algorithms operate on integers in unary representation, denoted by 0, $S(0)$, and so on, where S stands for successor. Consider the following two algorithms for integer addition [20]:

$$add1(0,y) = y$$
$$add1(S(x),y) = S(add1(x,y))$$

$$add2(x,0) = x$$
$$add2(x,S(y)) = S(add2(x,y))$$

The standard extensional denotational semantics for $add1$, $add2$ maps them both into the addition function of type $N^2 \to N$, where N is the flat domain of natural numbers. A simple intensional semantics may be provided by using the lazy natural numbers [19, 20, 22]. The domain $LNAT$ is shown in Figure 1.1. $LNAT$ captures the temporal aspect of finding out what an input is. At $S^k(\perp)$ we don't know yet if we have the number $S^k(0)$, or something larger (at least $S^{k+1}(\perp)$). This intensional semantics is sufficient to distinguish between the two addition algorithms. Using the meaning function $[\![\ ]\!]$ from [20, 22] (which makes the meaning $\perp$ when an algorithm tries to recur on $\perp$) we have:

$$[\![add1]\!](S^2(\perp), S(\perp)) = S^2(\perp)$$
$$[\![add2]\!](S^2(\perp), S(\perp)) = S(\perp)$$

The $LNAT$ semantics is richer than the N semantics, and contains intensional information; the above equations can be interpreted as showing that at some point, $add2$ tries to evaluate part of its second argument before the first, whereas $add1$ looks at its first input first. Although the $LNAT$ semantics still represents the meanings of $add1$ and $add2$ as functions (from $LNAT \times LNAT$ to $LNAT$), it conveys implicit information about the computation strategy of the related functions from N^2 to N.

Colson used the $LNAT$ semantics to study the efficient expressibility of a function that computes the *minimum* of two natural numbers represented in unary notation. Although the semantics appears quite simple, it was enough to allow Colson to prove a rather surprising impossibility result: $\mathcal{PR}$ algorithms cannot compute minimum efficiently. We shall present his work in more detail in the next chapter. The $LNAT$ semantics will also appear in our work, when we conduct our own study of the efficient expressibility of a minimum function in the context of sequential algorithms on concrete data structures, and their generalization to parallel algorithms.

1.2 Relative intensional expressiveness

In the first half of this thesis, we are interested in establishing relative intensional expressiveness results for programming languages. Most work in the past has focused on extensional expressiveness: Language L_1 is *extensionally more expressive* than L_2 if L_1 can compute all the functions computable in L_2. We say that language L_1 is *intensionally more expressive* than L_2, if L_1 can compute all the functions computable in L_2, with at least the same asymptotic complexity. The notions of complexity we concentrate on are time and work. Note that there has been a lot of work comparing the intensional expressiveness of different *models* of computation. For instance, allowing only a single tape for a Turing machine can square the time necessary to recognize a language versus a two-tape Turing machine [48]; and there are certain problems for which there exist faster CRCW PRAM algorithms than EREW PRAM algorithms [23]. Our goal is to compare programming languages, not their underlying computation models. We shall be careful to point out when we need to make special assumptions about the computation model in order to achieve our programming language comparisons.

It would appear that there should be close connections between relative intensional expressiveness and complexity theory. Indeed, there has been some work on machine-independent characterizations of complexity classes. A long time ago, Cobham [18] characterized P as a language similar to primitive recursive algorithms. Very recently, Clote [17] did the same for NC, which can be viewed as the class of functions that can be computed "quickly" in parallel. The characterization of NC also takes the form of a variant of primitive recursive algorithms. One of the main problems of complexity theory, P versus NC, can then be viewed as a problem of relative intensional expressiveness.

Quite obviously, we should not expect the act of viewing a problem as a relative intensional expressiveness problem on programming languages to make it easier. Proving negative results and lower bounds is difficult, no matter how one looks at it. It should be interesting to see, however, if any useful new ideas emerge at the interface of programming languages theory and complexity theory. We hope that our work can be seen as a small step in this direction.

We compare both sequential and parallel languages. First we examine the efficient expressibility of minimum on lazy natural numbers in CDS0 and in a parallel extension of CDS0 we call CDSP, and contrast that to Colson's results with $\mathcal{PR}$ algorithms. Then we compare four deterministic parallel extensions of PCF [70], which is the prototypical sequential functional language. Finally, we compare a deterministic and a nondeterministic extension of PCF. To aid us in the comparisons of PCF extensions, we introduce a new intensional semantics called *circuit semantics*. Circuit semantics associates a gate with each basic construct of the language, and takes the meaning of a program to be a circuit. The dimensions of the circuit enable reasoning about running time and work required for execution. Circuit semantics also allows us to formalize a connection between deterministic and nondeterministic parallel PCF programs, and monotone and De Morgan boolean circuits [87], respectively.

1.3 Refinement types

The idea of refinement types is due to Freeman and Pfenning [35, 36]. In their work, the programmer may choose to decompose a type into a collection of subtypes by means of a recursive datatype declaration. The subtypes are called *refinements* of the original type. They also developed a type inference system which first obtains a regular type (not involving refinements) for a program, then attempts to obtain a refinement type for it by means of refinement type inference rules. The intended use for the system was as a programmer's aid in eliminating spurious warnings generated by the Standard ML type inference for missing patterns that were actually unreachable.

We think that the idea of refinement types is a very interesting one, but our approach has a very different flavor. We are interested in program analysis and we do not have refinement type inference rules. Instead we perform an abstract interpretation on the program directly (instead of doing it at the level of types). As in Freeman and Pfenning, the programmer has to specify refinements, and we rely on the program to have a regular type before trying to generate a refinement type for it. We shall have more to say about differences between the two systems later, when we present our approach in detail. For now, we wish to illustrate the basic idea with some examples.

Suppose we have a generic, lazy functional language with a syntax similar to that of Standard ML (the examples below are actual programs from our implementation). Suppose further that we decide to distinguish between *true* and *false*, *i.e.,* we want to refine *bool*. We would expect the following program with regular type $bool \rightarrow bool$:

```
val not = fn x => if x then false else true;
```

to have refinement type $true \rightarrow false \wedge false \rightarrow true$, where $\wedge$ denotes intersection of types. The intuitive meaning is that the program *not* has both types $true \rightarrow false$ and $false \rightarrow true$.

Something more interesting happens when we decide to refine a recursive type. Suppose we have integer lists (*intlist*), and we want to distinguish between empty lists (*empty_intlist*), lists with one element (*one_intlist*), and lists of two or more elements (*many_intlist*). Then we would expect the *map* function of regular type $(int \rightarrow int) \rightarrow intlist \rightarrow intlist$:

```
val map = letrec mapf =
            fn f => fn l => if null l then []
                            else (f (hd l)) :: ((mapf f) (tl l))
          in mapf
          end;
```

to have the following refinement type:

$$
\begin{aligned}
&(int \rightarrow int) \rightarrow empty\_intlist \rightarrow empty\_intlist \wedge \\
&(int \rightarrow int) \rightarrow one\_intlist \rightarrow one\_intlist \wedge \\
&(int \rightarrow int) \rightarrow many\_intlist \rightarrow many\_intlist.
\end{aligned}
$$

We make use of intensional semantics to help generate such refinement types by translating the programs to *categorical combinators* [26], which themselves denote sequential algorithms (*i.e.,* CDS0 programs). The types are represented as concrete data structures [56]. A type and its refinements will always be distinguishable by examination at a finite number of points. So we perform abstract interpretation of the CDS0 program over the lattice of such points, and use the very precise information on the dependence of *parts* of the output on *parts* of the input provided by sequential algorithms to generate the refinement type.

1.4 Claims of the thesis

The guiding principle behind this work and the central claim of the thesis is:

> The exploration of intensional semantics is interesting from both a theoretical and practical point of view.

More specifically, we claim the following:

- Definition of the notion of relative intensional expressiveness for programming languages.
- Definition of CDSP, a parallel extension of CDS0 with a *query* construct [12].
- Proof that CDS0 is more expressive than $\mathcal{PR}$ algorithms, but less expressive than CDSP.
- Formalization of a new intensional semantics, circuit semantics, and comparisons with parallel evaluation strategies [49]: call-by-speculation, parallel call-by-value, and parallel eager evaluation.
- Identification of a hierarchy of intensional expressiveness for deterministic parallel extensions of PCF: parallel conditional on booleans is equivalent to parallel or; both are less expressive than parallel conditional on integers, which in turn is less expressive than *query*.
- Formalization of a connection between work and time complexity of functional programs extended with deterministic and nondeterministic *query*, and monotone and De Morgan circuits. Use of this connection and a hardware assumption to show that nondeterministic query is more expressive than the deterministic one.
- Development of a type system based on concrete data structures. Implementation of type inference for CDS0.
- Proof of soundness for both type inference and refinement type inference.
- Development of a new application of CDS0.
- Implementation of a practical approach to refinement type inference.

1.5 Related work

We consider separately related work in the areas of intensional semantics, relative intensional expressiveness, and refinement types and type inference for CDS0.

1.5.1 Intensional semantics

The related work surveyed here is composed of several different strands. The common element is a concern with the analysis of intensional aspects of programs. In most cases, the programming language is sequential, and the analysis is carried out from an operational presentation of the semantics. The notable exceptions will be pointed out.

We discuss relevant work on the following topics: intensional semantic models for programming languages, intensional hierarchies, and automatic complexity analysis. The section is broken down by area (*e.g.,* recursion theory), rather than topic, to give an idea of the naturality and pervasiveness of these ideas.

Recursion theory

The distinction between a function and the algorithm that computes it was made early on in recursion theory [76], but the main focus of the theory is on the functions, that is on the extensional features. Most results have to do with closure properties of various collections of recursive functions. However, a so-called "abstract" recursion theory (also called theory of algorithms) has been developed, chiefly by Moschovakis [64, 65, 66], although the ideas go back to Kleene and others. In [64] Moschovakis develops the foundation for the theory. The semantics of a recursive partial function is a set of functionals, called a *recursor*. It is essentially a higher-order functional program defined by a family of mutually recursive function definitions. Intensional analysis can be performed in an operational style on the recursor. The possibility of implementing the language of recursors as a programming language called REC is discussed. More recent works [65, 66] update and elaborate on the older paper. Algorithms are modeled as recursors, which are part of a programming language called FLR (Formal Language of Recursion). The main thrust is in proving that FLR is a reasonable language in terms of including all desirable intensions.

Proof theory

Proof theory [38, 39] has been mainly concerned with extensional aspects, as well. A series of functional systems of increasing *extensional* expressive power has been studied: linear λ-calculus, typed λ-calculus, primitive recursion, Gödel's system T, Martin-Löf's intuitionistic type theory, Girard-Reynolds polymorphic second-order λ-calculus (system F), and the theory of constructions. None of these systems is Turing-complete; all their programs terminate.

Recently there has been work on intensional aspects of some of these functional systems. Colson's work [19, 20] with primitive recursive algorithms was mentioned already. He also studied system T and system F. System T can express an efficient algorithm for minimum. It is an open problem whether $min(n,p)$ can be written in system F with complexity $O(min(n,p))$. The current best program (see [29]) is $O(min(n,p)log(min(n,p))$. Interestingly, system T appears to be intensionally stronger than system F, even though it is extensionally weaker.

Programming languages

There is a large body of literature devoted to automatic complexity analysis. There are typically two phases to an automatic complexity analysis system: deriving recurrences for the complexity of a program, and solving them. Deriving the recurrences is usually accomplished by constructing a cost (or complexity) function from the program and obtaining from this a function of the input size. The cost function normally counts the number of rewrites in the operational semantics, plus constants for the primitive operations.

Most of the work in automatic complexity analysis has been devoted to studying strict, sequential, first-order, functional languages (the earliest examples are Wegbreit [86] and Le Metayer [60]). There has also been some effort in the area of lazy first-order languages [79, 85]. The derivation of a program's complexity is more complicated in this setting, because only part of an argument might be needed. There has also been work with higher-order strict and lazy languages [78]. The basic idea is to construct *cost-closures* so a function can carry around cost information.

Several authors have used *profiling semantics*, *i.e.,* operational semantics augmented with time and work information, to perform automatic complexity analysis in a parallel setting. Roe [75] considered a parallel *lenient* language, and Zimmerman [89, 90] a data-parallel language. The language Zimmermann analyzes is a first-order parallel language with vectors and a parallel "forall" construct (and similar others) ranging over vectors. The approach is the same as in the sequential case: a cost function is constructed, with the cost of the parallel "for-all" equal to some constant plus the *maximum* cost of the operation over each vector element.

Hudak and Anderson [49] developed pomsets as a semantics for parallel functional programs. They were able to distinguish between various evaluation strategies (call-by-value, call-by-name, call-by-need, call-by-speculation). More recently, Blelloch and Greiner [9, 41] provided profiling semantics for parallel call-by-value and call-by-speculation. Their aim in [9] was to show that good upper bounds for merging and sorting can be obtained with an implicitly parallel language. The second model [41] was used to prove the efficiency of a particular implementation of call-by-speculation. Both models were related to more traditional parallel models such as the PRAM.

The circuit model we develop in this paper is most closely related to call-by-speculation. The differences are due to the presence of conditionals in the language. In contrast to earlier work, we are interested in proving lower bounds and performing intensional comparisons between parallel languages.

An interesting approach to automatic complexity analysis, in the setting of strict, sequential, first-order languages, was taken by Rosendahl [77]. He also constructs a time (or cost) function from the program, but in order to talk about the correctness of this time function, he defines an "instrumented" denotational semantics which returns a denotation and the time complexity. A *time-bound* function, which gives an upper bound on computation time for all inputs of a certain size, is derived by abstract interpretation from the time cost function.

Talcott developed a theory of intensional semantics [83]. Essentially, the extraction of the intensional information is based on a low-level operational semantics: from a program she constructs a *computation sequence*. Analysis is performed on the computation sequence: the time complexity of a program is the length of its computation sequence. Other properties can be analyzed, such as maximum stack depth and number of function calls.

Gurr [44] extended denotational semantics in order to model intensional aspects (resource requirements) of first-order, sequential languages. In his framework, the meaning of a program is a pair of the original denotation of the program and a map from input values to an object of resource values. The object of resource values is modeled as a *monoid* (a semigroup with identity). Time and space requirements of programs can be formulated in this framework. He also studied the derivation of exact and non-exact complexities.

1.5.2 Relative intensional expressiveness

The only example we are aware of which compared the intensional expressiveness of two programming languages is the already mentioned work of Colson on the expressiveness of primitive recursion.

There has been little work on comparing determinism and nondeterminism. Felleisen, in his theory of expressiveness [32], defined a construct c as more expressive than another c' if the translation of a program using c to one using c' requires a *global reorganization* of the program. He showed that adding side-effects to a sequential functional language increases expressive power.

The literature on Id [82], an *implicitly* parallel language, has produced practical examples of comparisons of Id's purely functional core, the extension with I-structures (single-assignment arrays), and the extension with M-structures (arrays with element-level synchronization).

There has been notable work on *extensional* comparisons of merging primitives in dataflow networks [68]. One of the functions considered there is *poll* which checks, without blocking, if an input is present. We make use of *poll* in our work. However, we are not aware of any relevant *intensional* comparisons of parallel constructs. The general opinion expressed in [62] (and echoed in [47]) seems to be that the main advantage of nondeterminism is in *specifying* a process.

There has been some recent research at the juncture of complexity theory and programming languages theory, with broadly the same aim of bridging the gap between these two areas of computer science. Aside from the machine-independent characterizations of P and NC already mentioned, there has been work on the characterization of P in terms of bounded linear logic [40] and λ-calculus [59]. In addition, Jones has commenced a reconstruction of computability and complexity theory from a programming languages perspective [53, 54].

1.5.3 Refinement types and type inference for CDS0

Pierce [69] introduced $F_\wedge$, a variant of system F with intersection types, subtyping, and bounded quantification. Using an explicit alternation construct called *for*, this system can derive refinement types. Unfortunately the system is too powerful; it has explicit types, and type checking is undecidable.

Reynolds developed the programming language Forsythe [74], which has intersection types and subtyping, but no polymorphism. In this system, an intersection type can contain a mixture of ground and higher-order types.

There has been much work on type systems using intersection types [46]. Such systems are usually too powerful to admit type inference; [21] is an exception. Fuh and Mishra [37] developed a type inference system which combines polymorphism and subtyping, but does not have intersection types.

There has also been a lot of work on type systems based on records (see [43] for several examples, including type inference systems). Concrete data structures (cds) are, in some sense, similar to records; they have *cells* (like fields in a record) which can be filled with *values*. In addition, however, cds have *accessibility* conditions, but this is not essential: the notion of subtyping we develop for cds is very similar to that for records. More important is the fact that a higher-order type in CDS0 is also a cds, and we can interactively ask questions about the values of its cells.

Soft typing [15] is a type inference system which includes polymorphism, subtyping, and union types, and it is designed for dynamically-typed languages; when a program fails to have a static type, run-time checks are included. The main thrust of this system is to be able to type programs that would normally be rejected by standard type checking.

Castagna, Ghelli, and Longo [16] defined the $\lambda\&$-calculus, a calculus for overloaded functions with subtyping. A function can be overloaded with the addition of new pieces of code. The types of the various pieces have some consistency conditions. In CDS0, programs can use generic cell and value references which can result in overloading. Our notion of overloading for CDS0 types is similar to [16], except that we do not build the consistency conditions into the type; we adopt the same notation.

1.6 Outline

Chapter 2 describes the work that we are building upon most directly in this thesis. We discuss the full abstraction problem for PCF and Kahn and Plotkin's definition of sequentiality using concrete data structures. This was the starting point of Berry and Curien's work on sequential algorithms on concrete data structures and its implementation as a programming language, CDS0. We spend a fair amount of time on CDS0 as it features prominently in the second part of this thesis. We describe Brookes and Geva's work on a parallel extension of CDS0, which provides us with one of the deterministic parallel constructs we study. Hughes and Ferguson's use of CDS0 to perform abstract interpretation is also covered. Although our approach is different, knowledge of the earlier work is useful. Finally, we provide a brief exposition of Freeman and Pfenning's work on refinement types, and Colson's work on intensional expressiveness.

Chapter 3 begins our relative intensional expressiveness explorations. Relying on Colson's work, we show that CDS0 is more expressive than $\mathcal{PR}$ algorithms. Even though CDS0 programs are sequential, they are not "ultimately obstinate," like the $\mathcal{PR}$ algorithms. However, CDS0 still cannot compute a natural version of the minimum function on lazy natural numbers. The parallel extension CDSP can compute that function. In addition, CDSP can compute certain n-ary functions more efficiently than CDS0.

Circuit semantics is introduced in Chapter 4. Initially, we introduce only a naïve version of circuit semantics that can essentially only distinguish programs based on depth. This is enough to compare four deterministic parallel extensions of PCF and separate them into three levels of intensional expressiveness. We then commence a more careful development of circuit semantics, comparing it to various parallel evaluation strategies, and using it to model a deterministic and nondeterministic parallel extension of PCF. We formalize a connection between the circuit dimensions of parallel PCF programs and monotone and De Morgan boolean circuits. Utilizing strong results from complexity theory, and assuming hardware that can detect undefined inputs, we are able to prove an intensional separation of the deterministic and nondeterministic construct.

Chapter 5 marks the beginning of the second part of the thesis. We carefully formalize a type system based on concrete data structures, that includes subtyping and intersection types. We show the decidability of subtyping for ground concrete data structures, and we introduce a type inference system, proving its soundness. Then we add polymorphism and overloading to the language, showing how to extend the subtyping decision procedure and the type inference system. We prove soundness for the extended system.

Refinement type inference is presented in Chapter 6. We define refinement types for CDS0 and show how the intensional information present in a sequential algorithm can be used to extract a refinement type. We introduce a generic, lazy functional language, and show how it can be compiled to CDS0. To derive refinement types for expressions built up from sequential algorithms, we introduce a loop-detecting evaluator, and show how we need only evaluate the expression at a certain (small) number of cells. We prove soundness of the refinement type inference.

Chapter 7 describes our prototype implementation and includes more examples. We outline briefly the implementation of CDS0 itself, which is based upon the work of Devin. Most of the chapter is devoted to our implementation of type inference and refinement type inference. We provide examples showing the benefits and limitations of our approach.

Finally, Chapter 8 looks back on the thesis drawing some conclusions and outlines areas for possible future work.

Chapter 2

Background

This chapter surveys the work that we will be building upon most directly in what follows. Section 2.1 gives a brief history of the full abstraction problem for PCF. Section 2.2 discusses Kahn and Plotkin's concrete data structures and their formulation of a notion of sequentiality, and its use by Berry and Curien in constructing an intensional semantics of sequential algorithms for PCF. Berry and Curien's programming language CDS0, which is a direct implementation of their intensional semantics, is described in Section 2.3. In Section 2.4 we describe Brookes and Geva's extension of Berry and Curien's work to the setting of parallel algorithms. Section 2.5 covers the only previous work on practical applications of sequential algorithms. Freeman and Pfenning's refinement type inference system is described in Section 2.6. Finally, Colson's work on the intensional expressiveness of primitive recursive algorithms is discussed in Section 2.7.

2.1 PCF and full abstraction

When a language is designed, the semantics which is normally regarded as "the definition" of the language is often presented in an operational style, by reference to the computations of an abstract machine, or, in the case of structural operational semantics [71], by a set of rewrite rules. This leads to a notion of program equivalence based on observability: two programs will be considered equivalent if, when inserted into the same *context* (intuitively, a program with a hole in it), we get the same final result after execution, as characterized by the abstract machine or by application of the rewrite rules.

If we give the language a denotational semantics as well, we obtain a different notion of program equivalence: two programs are equivalent if they denote the same value. It would be useful if these two notions of equivalence were identical: proving denotational equivalence would then automatically imply operational equivalence, and vice versa. If that were the case we would say that the denotational semantics is *fully abstract* with respect to the operational semantics [70]. The formulation is worded this way because the operational semantics is considered as intuitively known.

Unfortunately, it turns out to be rather difficult, in general, to design fully abstract denotational semantics [63, 8]. In the setting of sequential languages, there has been a great deal of effort expended in discovering a fully abstract semantics for the language PCF (Programming Computable Functions) [70, 42]. PCF is regarded as the "prototypical" sequential programming language. It is a typed λ-calculus with two ground types, booleans (o) and integers (ι) and the following set of

$tt, ff : o$	(truth values)
$n : \iota$	(integers, $n \geq 0$)
$isZero? : \iota \to o$	
$+1, -1 : \iota \to \iota$	
$\supset_o : o \to o \to o \to o$	(boolean conditional)
$\supset_\iota : o \to \iota \to \iota \to \iota$	(integer conditional)
$Y_\sigma : (\sigma \to \sigma) \to \sigma$	(one for each σ)

Figure 2.1: PCF constants

$\supset_\sigma tt\ M_\sigma N_\sigma \to M_\sigma, \quad \text{for } \sigma = \iota, o$

$+1\ n \to n+1, \quad \text{for } n \geq 0$

$\supset_\sigma ff\ M_\sigma N_\sigma \to N_\sigma, \quad \text{for } \sigma = \iota, o$

$-1\ (n+1) \to n, \quad \text{for } n \geq 0$

$Y_\sigma M \to M(Y_\sigma M)$

$isZero?\ 0 \to tt$

$((\lambda x.\ M)N) \to [N/x]M$

$isZero?\ (n+1) \to ff, \quad \text{for } n \geq 0$

$$\frac{M \to M'}{(MN) \to (M'N)}$$

$$\frac{N \to N'}{(MN) \to (MN')} \quad \text{if } M \text{ is } +1, -1, isZero?$$

$$\frac{M_0 \to M_0'}{(\supset_\sigma M_0) \to (\supset_\sigma M_0')}$$

Figure 2.2: Operational semantics for call-by-name evaluation of PCF

types, σ:

$$\sigma ::= o \mid \iota \mid \sigma \to \sigma$$

The syntax for raw (untyped) terms is given by the grammar:

$$M ::= c \mid x \mid \lambda x.\ M \mid MM$$

The constants traditionally included in the language are shown in Figure 2.1. The operational semantics for call-by-name evaluation is shown in Figure 2.2. For simplicity, we blur the distinction between numerals and integers, and use n to denote both. We omit the typing rules, which are standard.

The standard denotational semantics for PCF is given by the semantic function

$$\mathcal{D}: \text{Terms} \to \text{Environments} \to \bigcup D_\sigma,$$

where D_σ is a family of domains which includes the flat domains of booleans (D_{bool}) and integers (D_{int}), and such that $D_{\tau_1 \to \tau_2} = [D_{\tau_1} \to D_{\tau_2}]$, the continuous function space (see [70] for details).

Plotkin [70] showed that the standard denotational semantics $\mathcal{D}$ for PCF is not fully abstract. The problem is that the denotational semantics is "finer" than the operational semantics. It makes too many distinctions. If two programs are denotationally equivalent, then they are operationally equivalent. The converse does not hold. This happens because the denotational semantics contains functions which are not definable in the language, like parallel-or (*por*). *Por* returns true if at least one of its arguments is true:

$$
\begin{aligned}
&por\ tt\ \bot = tt\\
&por\ \bot\ tt = tt\\
&por\ \mathit{ff}\ \mathit{ff} = \mathit{ff}
\end{aligned}
$$

Using *por* we can construct a program context that distinguishes denotationally between two operationally equivalent programs. First, let $\Omega_\tau \equiv Y_\tau(\lambda f.\ f)$. Then, let

$$
\begin{aligned}
\text{ORTEST} \equiv \lambda i.\ \lambda f.\ \supset_\iota\ &(f\ tt\ \Omega_o)\\
&(\supset_\iota\ (f\ \Omega_o\ tt)\\
&\quad (\supset_\iota\ (f\ \mathit{ff}\,\mathit{ff})\ \Omega_\iota\ i)\\
&\quad \Omega_\iota\\
&)\\
&\Omega_\iota
\end{aligned}
$$

Now, ORTEST 0 is operationally equivalent to ORTEST 1, namely they both diverge. However, according to the denotational semantics,

$$\mathcal{D}[\![\text{ORTEST}]\!]\bot = \lambda v.\ \lambda\Phi.\ \textbf{if}\ \Phi = por\ \textbf{then}\ v\ \textbf{else}\ \bot,$$

and we have $\mathcal{D}[\![\text{ORTEST }0]\!] \neq \mathcal{D}[\![\text{ORTEST }1]\!]$.

Attempts to solve this problem have been aimed at eliminating the unwanted functions from the semantics, by restricting the continuous functions to "sequential" functions. Other ways of solving this are to change the operational semantics or to add primitives to PCF. Plotkin [70] showed that the denotational semantics for PCF + *por* (referred to as PCFP) is fully abstract. Cartwright, Curien, and Felleisen showed [14, 26] that full abstraction can also be obtained by extending PCF with a *catch* primitive (referred to as PCFC).

Recently, the full abstraction problem for PCF has been solved, independently, by several groups [2, 52, 67]. Interestingly, all these solutions are constructed in a similar way: First, an intensional semantics based on *game semantics* is constructed. Then, the undesirable intensional elements are filtered out, leaving behind an extensional model. The most fascinating part is that the game semantics can be seen as an elegant generalization of the work discussed in the next section, and upon which the language CDS0 is based.

2.2 Concrete data structures

Some of the early efforts at defining sequential functions were hampered by working in a setting where no distinction was made between function domains and domains of the data they compute on. In part to alleviate this, by providing a model in which it is easier to formalize the notion of incremental computation, Kahn and Plotkin [56] developed concrete data structures, and their domain-theoretic version, the concrete domains.

A concrete data structure is like a variant record in Pascal. It consists of a set of named *cells*, which can hold *values*, and an *accessibility* relation governing the order in which the cells can be *filled* with values. A cell filled with a value is called an *event*, written $c = v$. The following definitions are adapted from Curien [26].

Definition 2.2.1 *A concrete data structure (cds) is a tuple $(C, V, E, \vdash)$, where C, V, E are sets of cells, values and events, respectively, such that*

$$E \subseteq C \times V \text{ and } \forall c \in C, \exists v \in V.\ (c, v) \in E$$

and $\vdash$ is a relation, called an accessibility relation, between finite subsets of E and elements of C. We say that $\{e_1, \ldots, e_n\}$ is an enabling of c if $\{e_1, \ldots, e_n\} \vdash c$, which may also be written $e_1, \ldots, e_n \vdash c$. A cell such that $\emptyset \vdash c$ (which is abbreviated $\vdash c$) is called initial.

Definition 2.2.2 *A state is a subset x of E such that:*

1. $(c, v_1), (c, v_2) \in x \Rightarrow v_1 = v_2$
   - *(no cell is filled more than once). This is called consistency.*
2. *If $(c, v) \in x$ then there exists a sequence of events $e_1, \ldots, e_n$ such that $e_n = (c, v), e_i = (c_i, v_i) \in x$, and $\{e_j \mid j < i\}$ contains an enabling of c_i for all $i \leq n$*
   - *(only enabled cells may be filled). This is called safety.*

The set of states of a cds M ordered by set inclusion is a partial order $\langle D(M), \subseteq \rangle$ called a *concrete domain*. If a domain D is isomorphic to $D(M)$, we say that M generates D.

Definition 2.2.3 *Given a state x of a cds, we say that a cell c is:*

- *filled (with v) in x if $(c, v) \in x$,*
- *enabled in x if x contains an enabling of c,*
- *accessible from x if it is enabled but not filled in x.*

The sets of filled, enabled, and accessible cells of x are denoted $F(x), E(x)$, and $A(x)$, respectively.

Definition 2.2.4 *A state y is said to cover a state x, written $x \prec y$, if $x < y$ and $\forall z. x < z \leq y$ we have $z = y$. In addition, we write $x <_c y (x \prec_c y)$ if $c \in A(x), c \in F(y)$ and $x < y (x \prec y)$.*

Definition 2.2.5 *A cds $M = (C, V, E, \vdash)$ is well-founded if the transitive closure of the relation $\ll$ defined on C by:*

$$c_1 \ll c \text{ if and only if an enabling of } c \text{ contains an event } (c_1, v)$$

is well-founded, i.e., *there is no infinite descending sequence $\ldots c_{n+1} \ll c_n \ll \ldots \ll c$.*

Definition 2.2.6 *A cds is called stable if for each state x and cell c enabled in x, c has a unique enabling in x.*

We are only interested in well-founded and stable cds in the sequel. We call such cds deterministic (dcds). In addition, we shall be using an operational semantics for CDS0 called CDS02 [7, 30], which requires the dcds to be sequential.

Definition 2.2.7 *A cds M is called sequential if, for every cell d, and each state x of M such that*

$$d \notin F(x) \text{ and } \exists y \geq x.\ d \in F(y),$$

there exists a cell c such that:

$$c \in A(x) \text{ and } \forall y \geq x, d \in F(y) \Rightarrow c \in F(y).$$

Such a cell c is called a sequentiality index of M for d at x.

A cds M such that all its enablings contain at most one event is called *filiform*. Filiform cds's are particularly well-behaved, and they make the presentation of some definitions much simpler.

Example 2.2.8 *We can define the dcds of booleans (BOOL) the following way: there is one cell called B, which can be filled with either tt or ff. The set of states of this dcds is:*

$$\{\{\}, \{B = tt\}, \{B = \mathit{ff}\}\}.$$

Note that $\langle D(BOOL), \subseteq \rangle$ is isomorphic to D_{bool}, the flat domain of booleans, i.e., *BOOL generates D_{bool}.*

Example 2.2.9 *The dcds of integers INT, can be defined in a similar fashion: there is one cell called N which can be filled with any integer value. Again, note that INT generates D_{int}, the flat domain of integers.*

Example 2.2.10 *We provide an example of a non-sequential dcds, which will become relevant when we present the operational semantics CDS02. The dcds is called STABLE and has four cells: B_1, B_2, B_3, C. The cells B_i are all initial with possible values tt, ff. Cell C has any integer as a possible value, and the following access conditions:*

$$\{B_1 = tt, B_2 = \mathit{ff}\} \vdash C$$
$$\{B_2 = tt, B_3 = \mathit{ff}\} \vdash C$$
$$\{B_3 = tt, B_1 = \mathit{ff}\} \vdash C.$$

The reason STABLE fails to be sequential is that we cannot determine sequentially if C is accessible; each of the access conditions omits one of the B_i cells, so we wouldn't know where to start to figure out if C is accessible. It lacks a sequentiality index.

We can view a product of two cds's as being composed of two sides: a left and a right hand side. The product is created by tagging the left hand side cds cells with a 1 and the right hand side cells with a 2.

Definition 2.2.11 *Let M and M' be two cds's. The product $M \times M' = (C, V, E, \vdash)$ is defined as follows:*

- $C = \{c.1 \mid c \in C_M\} \bigcup \{c'.2 \mid c' \in C_{M'}\}$,
- $V = V_M \bigcup V_{M'}$,
- $E = \{(c.1, v) \mid (c, v) \in E_M\} \bigcup \{(c'.2, v') \mid (c', v') \in E_{M'}\}$,
- $(c_1.1, v_1), \ldots, (c_n.1, v_n) \vdash c.1$ *if* $(c_1, v_1), \ldots, (c_n, v_n) \vdash c$ *(and similarly for the right hand side).*

It is easier to visualize the product of two cds's when they are both filiform. Table 2.1 summarizes the procedure.

	M	M'	$M \times M'$
Cells	c	c'	$c.1,\ c'.2$
Values	v	v'	$v,\ v'$
Events	(c,v)	(c',v')	$(c.1,v),\ (c'.2,v')$
Enablings	$(c_1,v_1) \vdash c$	$(c'_1,v'_1) \vdash c'$	$(c_1.1,v_1) \vdash c.1,\ (c'_1.2,v'_1) \vdash c'.2$

Table 2.1: Product of two filiform dcds's

2.2.1 Sequential functions

Using cds's, Kahn and Plotkin [56] defined a notion of sequential function.

Definition 2.2.12 *A continuous function f is sequential at some state x in its domain, if for each cell c' accessible in $f(x)$ either:*

1. *no cell is accessible in x, or*
2. *there is an accessible cell c that must be filled in any state y that is a superset of x such that c' is filled in $f(y)$. The cell c is called a* sequentiality index *of f at x for c'.*

A function is sequential if it is continuous and sequential at every x in its domain.

Intuitively, this definition captures the notion that a sequential function is at any point dependent on one of its inputs; if that input diverges, the function will diverge.

2.2.2 Sequential algorithms

Berry and Curien [6] showed that Kahn-Plotkin cds's and sequential functions do not form a cartesian closed category (ccc), hence they cannot be used to model PCF. However, Berry and Curien defined *sequential algorithms* on cds's, which do form a ccc. This model was not useful for solving full abstraction for PCF because it is intensional (and it is known that the solution must be extensional [63]), but that is exactly the feature of interest for this work; the meaning of a PCF term is an algorithm and the model is fully abstract with respect to a notion of observability that is sensitive to computation strategy. This is the first instance of an intensional model in the computer science literature of which we are aware.

Sequential algorithms can be viewed two ways: abstractly and concretely. Abstractly, a sequential algorithm is a pair of a sequential function and a (sequential) computation strategy. If there are several ways of proceeding during the computation, the computation strategy points out a particular one. Concretely, a sequential algorithm is a state of a dcds of arrow type (the *exponentiation* dcds).

Definition 2.2.13 *Given two dcds's, M and M', the exponentiation dcds $M \Rightarrow M'$ is defined [26] by:*

- *if x is a finite state of M and if c' is a cell of M', then xc' is a cell of $M \Rightarrow M'$.*
- *the values and events are of two types:*
    1. *type "valof": if c is a cell of M then **valof c** is a value of $M \Rightarrow M'$, and $(xc', \textbf{valof c})$ is an event of $M \Rightarrow M'$ if c is accessible from x,*

	M	M'	$M \Rightarrow M'$
Cells	c	c'	xc', where $x \in D(M)$
Values	v	v'	$\mathit{valof}\, c, \mathit{output}\ v'$
Events	(c, v)	(c', v')	$(xc', \mathit{valof}\, c)$, where $c \in A(x)$, $(xc', \mathit{output}\ v')$
Enablings	$(c_1, v_1) \vdash c$	$(c'_1, v'_1) \vdash c'$	$(yc', \mathit{valof}\, c) \vdash xc$, if $y \prec_c x$, $(xc'_1, \mathit{output}\ v'_1) \vdash xc'$

Table 2.2: Exponentiation of two filiform dcds's

2. *type "output": if v' is a value of M' then* $\boldsymbol{output\ v'}$ *is a value of $M \Rightarrow M'$, and $(xc', \boldsymbol{output\ v'})$ is an event of $M \Rightarrow M'$ if (c', v') is an event of M'.*

- *the enablings are also of two types:*
  1. $(yc', \boldsymbol{valof\ c}) \vdash xc'$ *if $y \prec_c x$ (type "valof"),*
  2. $(x_1c'_1, \boldsymbol{output\ v'_1}), \ldots, (x_nc'_n, \boldsymbol{output\ v'_n}) \vdash xc'$
     if $x = \bigcup\{x_i \mid i \leq n\}$ and $(c'_1, v'_1), \ldots, (c'_n, v'_n) \vdash c'$ (type "output").

A state of $M \Rightarrow M'$ is called a sequential algorithm.

Again, for ease of reading we provide a description of the procedure for constructing an exponentiation dcds for filiform dcds in Table 2.2.

Example 2.2.14 *The state of $BOOL \Rightarrow BOOL$ that corresponds to the boolean negation is:*

$$\{\{\}B = \mathit{valof}\ B, \{B = tt\}B = \mathit{output}\ \mathit{ff}, \{B = \mathit{ff}\}B = \mathit{output}\ tt\}.$$

The way to read this definition is: Given no information about the input and having to fill the output cell B, we ask what value the input cell B holds. If the input is true we output false and conversely.

Berry and Curien [6] defined application, composition, product, pairing, currying, uncurrying, and fixpoint for sequential algorithms, and showed that sequential algorithms and cds's form a ccc. They used sequential algorithms to construct an (intensional) model of typed λ-calculus with recursion; λ-expressions are translated to categorical combinators, which are represented by sequential algorithms.

2.3 The language CDS0

The programming language CDS0 [5, 6, 7, 30] is a direct implementation of the intensional denotational semantics presented above; hence, it is an intensional programming language of sequential algorithms. The name stands for Concrete Data Structures. The initial idea [5] was to make CDS0 a kind of "assembly" language for a (syntax-wise) ML-like language called CDS. Programs in CDS would be compiled down to CDS0. Only CDS0 was ever implemented [30].

CDS0 is a lazy, polymorphic, higher-order, functional language with several quite interesting features:

- Uniformity of types. Everything in CDS0 is a state of a dcds. This can be a state-constant or a higher-order algorithm. The algorithm syntax is just syntactic sugar for the state of a dcds. Consequently, an algorithm can be evaluated without being applied to any argument. Operationally speaking, terms of non-ground type can be observed.

- Full abstraction. The denotational semantics of CDS0, which maps an algorithm to a state of the dcds corresponding to its type (hence a CDS0 object) is fully abstract with respect to two different operational semantics (CDS01 and CDS02) [26]. Since the semantics of an algorithm is a CDS0 object it is possible to write algorithms which manipulate the *semantics* of other algorithms.

- Demand-driven, coroutine-like evaluation style. The user types in an expression and enters a *request* loop, where questions about various cells of the dcds that is the type of the expression can be asked. If the expression is a state-constant the value of the cell is simply looked up in the state; if the expression is compound, processes are associated with each subexpression and they exchange information while computing the value of the cell. The computation style is an extension of the *coroutine* mechanism of Kahn and MacQueen [55].

- Rich data structure definition facilities. Cds's are very general and permit definition of a wide variety of data structures. In particular, they can be defined recursively, and one cds can be *grafted* into another. A thorough discussion of the type definitions in CDS0 will be deferred to the second part of this thesis, where we develop a type inference system for CDS0.

The examples in this section are from our own implementation of CDS0, which follows that of Devin [30] for the untyped part. We shall describe our implementation in more detail in Chapter 7.

2.3.1 Type definitions

The types in CDS0 are dcds's. Only ground dcds's can be defined; higher-order ones must be created out of pre-existing dcds's. We begin by defining the dcds's we have already encountered in the previous section:

```
let bool = dcds
  cell B values tt, ff
end;

let int = dcds
  cell N values [..]
end;

let stable = dcds
  cell B1 values tt, ff
  cell B2 values tt, ff
  cell B3 values tt, ff
  cell C values [..] access B1=tt,B2=ff or B2=tt,B3=ff or B3=tt,B1=ff
end;
```

The *graft* construct allows us to copy an already defined dcds into another, tagging all its cells with a specified tag and optionally adding accessibility conditions. Here is a simple example:

```
let tagged_bool = dcds
  graft (bool.foo)
end;
```

This declaration creates a dcds with a single cell, $B.foo$ with possible values tt, ff. Grafting is more useful when used in conjunction with recursive declarations. We could define a stream of integers in the following fashion:

```
letrec int_stream = dcds
  cell (N.l) values [..]
  graft (int_stream.l) access (N.l) = [..]
end;
```

We have created an infinite dcds with cells of the form $N.l, N.l.l, \ldots$, each of which can have any integer as a value, and such that one cell has to be filled in order for the next one to become enabled. We can explore in our interpreter the structure of this dcds, by asking it to unroll the dcds:

```
# show more 3 int_stream;
{
(N.l) values [..],
((N.l).l) values [..] access (N.l)=[..],
(((N.l).l).l) values [..] access ((N.l).l)=[..]}
```

Cell names that become ever-longer as a dcds is being unrolled are a typical feature of CDS0 recursive type declarations.

2.3.2 Interaction with the interpreter

The states we encountered in the previous section can be typed "as is" into the interpreter. We begin with a state of $BOOL$. We omit typing considerations.

```
# {B = tt};
request? B;
--> tt
request? ;
#
```

Note how we entered the request loop, and examined the contents of cell B. Unsurprisingly, it was filled with tt. We can perform the same kind of examination of a higher-order state, *i.e.*, we can explore an algorithm without applying it to an argument. Here is boolean negation again:

```
# { {}B = valof B, {B=tt}B = output ff, {B=ff}B = output tt };
request? {}B;
--> valof B
```

Already, we have some idea of the computation strategy of this algorithm: we know it examines its input. By continuing the questions and answers further, we can find out what it does with its two possible inputs:

```
request? {B=tt}B;
--> output ff
request? {B=ff}B;
--> output tt
```

2.3.3 Algorithm syntax

Writing algorithms solely in state form would quickly become tedious, so an alternative syntax is provided. It is important to realize, however, that the notation is just syntactic sugar for a state. An algorithm from M to M' will have the general form:

```
algo
  request c1' do
    <instruction>
  end
  request c2' do
    <instruction>
  end
  ...
end
```

The algorithm contains a number of *request-do* branches, each of which specifies a recipe for computing the value of an output cell c_i'. There are two kinds of instructions:

```
output v'
```

which outputs a value into a cell of M' and

```
valof c is
  v1 : <instruction>
  ...
  vn : <instruction>
end
```

which tests a certain input cell and branches accordingly.

If an output cell c' is not initial we must specify, after the request-do construct, how it can become enabled. This is done with a *from-do* construct:

```
from <input state 1> do
  <instruction>
end
from <input state 2> do
  <instruction>
end
...
```

Two simple algorithms that work on booleans are shown in Figure 2.3. The first is the boolean negation, which we have encountered already in state form. The two forms are equivalent (they actually map into the same internal representation, as we shall see). The second is "left and," which performs a boolean conjunction, testing its left input first; if that input is *ff* it outputs the value *ff* right away, so it is not strict in both arguments. We can actually write four different

```
let not =
  algo
    request B do
      valof B is
        tt : output ff
        ff : output tt
      end
    end
  end;
```

```
let land =
  algo
    request B do
      valof (B.1) is
        tt: valof (B.2) is
              tt: output tt
              ff: output ff
            end
        ff: output ff
      end
    end
  end;
```

Figure 2.3: Boolean negation and left conjunction algorithms

kinds of boolean conjunction algorithms: "left and," "left strict and" (which would check its right argument even if the left one is *ff*), "right and," "right strict and."

An example of an algorithm which uses the *from-do* construct is shown in Figure 2.4. The algorithm acts like the identity on the initial cells of *STABLE*, but then distinguishes how cell C became enabled.

2.3.4 Polymorphism

Polymorphism arises in CDS0 through the use of generic (*i.e.*, variable) cell and value names. Variable names start with the special symbol "$". For example, this is how we could write the polymorphic identity:

```
let id = algo
    request $C do
        valof $C is
            $V : output $V
        end
    end
end;
```

Note that the $\$C$ from the output name is the same as the input one, and similarly for the values. The output cell name gets bound to a non-variable name first, when a query is issued. When the answer is returned, the input value gets bound to a non-variable first, then is copied to the output value. To make this clearer, we evaluate *id* by itself:

```
# id;
request? {}B;
--> valof B
request? {B=tt}B;
--> output tt
```

```
let distinguish =
  algo
    request B1 do
      valof B1 is
        tt : output tt
        ff : output ff
      end
    end
    request B2 do
      valof B2 is
        tt : output tt
        ff : output ff
      end
    end
    request B3 do
      valof B3 is
        tt : output tt
        ff : output ff
      end
    end
    request C do
      from {B1=tt, B2=ff} do
        output 1
      end
      from {B2=tt, B3=ff} do
        output 2
      end
      from {B3=tt, B1=ff} do
        output 3
      end
    end
  end;
```

Figure 2.4: Algorithm utilizing *from-do* construct

Combinator	*Syntax*	*Argument Type*	*Result Type*
Application	$A.B$	$A: D \to D'$, $B: D$	$A.B: D'$
Composition	$A \vert B$	$A: D' \to D''$, $B: D \to D'$	$A \vert B: D \to D''$
Fixpoint	$fix(A)$	$A: D \to D$	$fix(A): D$
Curry	$curry(A)$	$A: D \times D' \to D''$	$curry(A): D \to D' \to D''$
Uncurry	$uncurry(A)$	$A: D \to D' \to D''$	$uncurry(A): D \times D' \to D''$
Pair	$< A, B >$	$A: D \to D_1$, $B: D \to D_2$	$< A, B >: D \to D_1 \times D_2$
Product	(A, B)	$A: D_1$, $B: D_2$	$(A, B): D_1 \times D_2$

Table 2.3: CDS0 combinators

Since id has type $\forall\alpha.\ \alpha \to \alpha$, its cells will be higher-order. First we asked what the output cell B is, given no information about the input, and id replied that it needs to know the value of the eponymous input cell. When given that value, it simply copied it over to the output.

2.3.5 Categorical combinators

Sequential algorithms and states can be combined to form expressions using the categorical combinators. There are seven combinators: application ("."), composition ("|"), fixpoint ("fix"), curry ("$curry$"), uncurry ("$uncurry$"), pair ("< , >"), and product ("(,)"). For ease of reference, we list the combinators, along with their types in Table 2.3. The language of expressions is given by the following grammar, where x stands for a state-constant, and a for an algorithm declaration:

$$e ::= \ x \mid a \mid e.e \mid e|e \mid fix(e) \mid curry(e) \mid uncurry(e) \mid < e, e > \mid (e, e).$$

As an example of the use of combinators, we examine two simple expressions in the interpreter:

```
# not.{B=tt};
request? B;
--> ff
request? ;
# not | land;
request? {}B;
--> valof (B.1)
request? {(B.1)=ff}B;
--> output tt
```

2.3.6 Forest representation

Before presenting the operational semantics, we introduce the internal representation of *forests* for sequential algorithms developed by Devin [30]. Sequential algorithms already have a tree structure, so forests are quite similar. The two differences are:

1. The lists of from-do instructions become a tree of *From* instructions, similar to valofs. This is possible when we restrict ourselves to sequential dcds's.

2. When asking the value of an input cell, we specify which input it comes from, *i.e.,* given a type $M_1 \to M_2 \to \cdots \to M_n$, when we perform a valof of cell c_i from M_i, this becomes a *Valof*

```
let curry_land =
  algo
    request {}B do
      valof B is
        tt: output valof B
        ff: output output ff
      end
    end
    request {B=tt}B do
      from {B=tt} do
        output output tt
      end
    end
    request {B=ff}B do
      from {B=tt} do
        output output ff
      end
    end
  end;
```

```
# print land;
land =
{B=valof ((B.1), 1) is
      tt : valof ((B.2), 1) is
          tt : output tt
          ff : output ff
      ff : output ff
}

# print curry_land;
curry_land =
{B=valof (B, 1) is
      tt : valof (B, 2) is
          tt : output output tt
          ff : output output ff
      ff : output output ff
}
```

Figure 2.5: Curried left conjunction and internal representations

(c_i, i) instruction. This has the effect of making currying and uncurrying a simple game on input cell indices and tags.

A forest contains several trees, each one containing an *instruction* specifying how to compute one output cell: $\mathit{Tree}\ (c_1, \mathit{instruction}_1), \ldots, \mathit{Tree}\ (c_n, \mathit{instruction}_n)$. The instructions are of three kinds:

$$\mathit{Result}\ \mathit{output}^n\ v \qquad \begin{array}{l} \mathit{Valof}\ (c,i)\ \mathit{is} \\ \quad v_1\ :\ \mathit{instruction}_1 \\ \quad \ldots \\ \mathit{end} \end{array} \qquad \begin{array}{l} \mathit{From}\ (c,i)\ \mathit{is} \\ \quad v_1\ :\ \mathit{instruction}_1 \\ \quad \ldots \\ \mathit{end} \end{array}$$

Figure 2.5 shows a sequential algorithm for the curried version of "left and" and the internal representation as a forest for this algorithm and the earlier presented "left and." Note that despite their very different syntax trees, the two algorithms map into very similar forests, with the only differences occurring in the number of inputs and the product tags on the input cells.

We are now ready to present the operational semantics for the evaluation of forests. (All the CDS0 operational semantics rules are summarized in Appendix A.1 for ease of reference.) The rules are of the form *forest* $?\ c\ \rightarrow\ v$. We introduce a special value, Ω, which stands for an unfilled cell. This is different from $\bot$: When attempting to find a cell's value, the interpreter may loop, in which case the value would be $\bot$. It is possible, however, that the interpreter can figure out that the cell is not filled in that state, in which case the result would be Ω. The first two rules specify the search of a forest for the proper tree.

$$(\textsc{Tree1}) \qquad \frac{c'_i = c'}{\mathit{Tree}\ (c'_1, \mathit{ins}_1), \ldots, \mathit{Tree}\ (c'_n, \mathit{ins}_n)\ ?\ x_1 \cdots x_n c'\ \rightarrow\ \mathit{ins}_i\ ?\ x_1 \cdots x_n c'}$$

$$(\textsc{Tree2}) \qquad \frac{\forall i.\ c'_i \neq c'}{\textit{Tree}\ (c'_1, ins_1), \ldots, \textit{Tree}\ (c'_n, ins_n)\ ?\ x_1 \cdots x_n c' \ \rightarrow\ \Omega}$$

When executing a *Result* instruction, we simply output the resultant value.

$$(\textsc{Result}) \qquad \textit{Result}\ v'\ ?\ x_1 \cdots x_n c' \ \rightarrow\ v'$$

A *Valof* (c_i, i) of a cell $x_1 \cdots x_n c'$ specifies that we need to ask the i'th state part of the cell name $x_1 \cdots x_n c'$ what value c_i has. If that sub-query returns a value we look up the appropriate branch of the *Valof*. If no branch matches we return Ω, and if the sub-query returns Ω we return an answer that specifies that we still need the value of cell c_i.

$$(\textsc{Valof}) \qquad \frac{x_p\ ?\ c \ \rightarrow \begin{cases} v_i \\ v, \ \text{and}\ \forall i.\ v_i \neq v \\ \Omega \end{cases}}{\left.\begin{array}{l} \textit{Valof}\ (c,p)\ is \\ \ \ v_1\ :\ ins_1 \\ \ \ \cdots \\ \textit{end} \end{array}\right\}\ ?\ x_1 \cdots x_n c' \ \rightarrow \begin{cases} ins_i\ ?\ x_1 \cdots x_n c' \\ \Omega \\ output^{p-1}\ valof\ c \end{cases}}$$

From instructions are very similar to *Valof*. When the sub-query returns a value not in the list we fail by raising an exception, because the *From* cannot be satisfied. If the sub-query returns Ω, we also return Ω without failing, because it might still be possible to increase the input state and potentially satisfy the *From*.

$$(\textsc{From}) \qquad \frac{x_p\ ?\ c \ \rightarrow \begin{cases} v_i \\ v, \ \text{and}\ \forall i.\ v_i \neq v \\ \Omega \end{cases}}{\left.\begin{array}{l} \textit{From}\ (c,p)\ is \\ \ \ v_1\ :\ ins_1 \\ \ \ \cdots \\ \textit{end} \end{array}\right\}\ ?\ x_1 \cdots x_n c' \ \rightarrow \begin{cases} ins_i\ ?\ x_1 \cdots x_n c' \\ \text{fail with no-access} \\ \Omega \end{cases}}$$

2.3.7 CDS02 operational semantics

The first operational semantics devised for CDS0 was called CDS01 [26], and it involved the use of *tables* to store temporary results when evaluating certain constructs. The reason for the tables is that in some cases it is impossible to re-derive the enablings that allowed us to reach a certain point. Consider applying the algorithm *distinguish* from Figure 2.4, of type $STABLE \rightarrow STABLE$, to a state of $STABLE$, let us call it arg, in which attempting to evaluate B_1 loops, but $B_2 = tt, B_3 = ff$. If we have already evaluated B_2, B_3, then $distinguish.arg\ ?\ C$ should return 2. But if we did not store the previously evaluated B_2, B_3, we would have no way of knowing how to go about finding C's value, and we would loop if we chose to evaluate B_1.

The tables in CDS01 were only necessary for application, composition, and fixpoint. Carrying previously evaluated values around led to efficient evaluation of fixpoints, but in general CDS01 was inefficient in both space usage and time, because of searches in the tables.

In CDS02 [30] the restriction is made that all dcds's be sequential, so $STABLE$ is outlawed and one no longer has to keep the tables. The restriction to sequential dcds's is not essential in practice since most usual data structures are sequential. CDS02 is slower in the evaluation of fixpoints, but is overall more efficient than CDS01.

It is possible to pose *intensional queries* in CDS02. Whereas in CDS01 states are always explicitly given by enumeration of events, in CDS02 the interpreter manipulates expressions. The rule for application, given below, illustrates the point.

$$(\textsc{App}) \qquad \frac{A \ ?\ Bc' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ c \\ \mathit{output}\ v' \end{cases}}{A.B \ ?\ c' \rightarrow \begin{cases} \Omega \\ \Omega \\ v' \end{cases}}$$

Instead of constructing approximations x to the state of B and posing queries of the form $A\ ?\ xc'$, as would be done in CDS01, we simply package the expression B with the cell c' and ask that question of A. This essentially leads to a direct dialog between "interested parties" rather than having it be centralized through the use of tables.

The rules for composition and fixpoint are in the same vein.

$$(\textsc{Comp}) \qquad \frac{A \ ?\ (B.x)c'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ c' \quad B?xc' \rightarrow \mathit{valof}\ c \\ \mathit{output}\ v'' \end{cases}}{A|B \ ?\ xc'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ c \\ \mathit{output}\ v' \end{cases}}$$

$$(\textsc{Fix}) \qquad \frac{A \ ?\ \mathit{fix}(A)c \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ c' \\ \mathit{output}\ v \end{cases}}{\mathit{fix}(A) \ ?\ c \rightarrow \begin{cases} \Omega \\ \Omega \\ v \end{cases}}$$

The rule for fixpoint is actually an optimization that builds in one application step of the following rule:

$$(\textsc{Fix'}) \qquad \mathit{fix}(A) \ ?\ c \rightarrow A.\mathit{fix}(A) \ ?\ c$$

The remaining rules are simple games on the product tags of a cell. In the rule for uncurry, π_1, π_2 are the first and second projections.

$$(\textsc{Pair}) \qquad < A_1, \ldots, A_n > \ ?\ x(c.i) \rightarrow A_i \ ?\ xc$$

$$(\textsc{Prod}) \qquad \prod_{i=1}^{n} A_i \ ?\ (c.i) \rightarrow A_i \ ?\ c$$

$$
(\textsc{Curry}) \qquad \frac{A \ ? \ (x \times y)c'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ (c.1) \\ \mathit{valof}\ (c'.2) \\ \mathit{output}\ v'' \end{cases}}{\mathit{curry}(A) \ ? \ xyc'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ c \\ \mathit{output}\ \mathit{valof}\ c' \\ \mathit{output}\ \mathit{output}\ v'' \end{cases}}
$$

$$
(\textsc{Uncurry}) \qquad \frac{A \ ? \ (\pi_1.x)(\pi_2.y)c'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ c \\ \mathit{output}\ \mathit{valof}\ c' \\ \mathit{output}\ \mathit{output}\ v'' \end{cases}}{\mathit{uncurry}(A) \ ? \ xc'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ (c.1) \\ \mathit{valof}\ (c'.2) \\ \mathit{output}\ v'' \end{cases}}
$$

2.3.8 Related languages

A language in some ways similar to CDS0 has recently been developed by Cartwright, Curien, and Felleisen [14]. It is called SPCF (Sequential PCF) and it extends PCF with error values and primitives for non-local transfer of control (catch and throw). This enables an SPCF program to observe and exploit order of evaluation in other programs. SPCF programs are called observably sequential algorithms. If there are no errors, SPCF collapses into CDS0. In the presence of errors, SPCF is an extension of CDS0.

2.4 Parallel algorithms on concrete data structures

Brookes and Geva [12] proceeded to extend Berry and Curien's work to the setting of deterministic parallel algorithms on concrete data structures. The aim was to provide a general intensional theory of deterministic parallel computation.

A parallel algorithm can also be viewed two ways. Abstractly, it is a pair of a *continuous* function and a (parallel) computation strategy. Concretely, it is a program in a language of parallel algorithms.

The key change to sequential algorithms to yield parallel ones is to replace the *valof* construct with a parallel *query* construct, which, intuitively, spawns off a number of *valofs*. More precisely, a *query* starts a number of parallel sub-computations and specifies conditions based on the results of the sub-computations under which the main computation may resume. As an example, here is how we could write parallel-or (of type $BOOL^2 \rightarrow BOOL$) in syntax meant to look like CDS0 as much as possible:

```
let por =
  algo
    request B do
      query {(B.1), (B.2)} is
        {tt, _}  : output tt
        {_, tt}  : output tt
        {ff, ff} : output ff
      end
    end
  end;
```

The previous algorithm, while deterministic, is a little misleading, because it seems to imply that one has knowledge of which argument evaluated, so one could write a non-deterministic program. To ensure determinism we have to make sure the output is the same for all *consistent* input states. Another problem is caused by higher-order parallel algorithms: their queries have to apply not only to the immediate input, but also possible further inputs. To account for all this, the notation would have to be somewhat different (see [12]).

Brookes and Geva were able to define application, currying and uncurrying of parallel algorithms. However, this only works for first-order types and composition was not defined. Thus, they did not obtain a ccc of parallel algorithms on concrete data structures.

It is important to note that Berry and Curien also had difficulty defining composition for sequential algorithms; their solution was to define it in terms of the abstract view of sequential algorithms.

2.5 Applications of sequential algorithms

A sequential algorithm contains detailed information about the relationship between input and output. It does not simply tell us how the output depends on the input, but precisely how *parts* of the output depend on *parts* of the input. Seizing on this intensional information, Hughes and Ferguson [33, 50] developed applications using sequential algorithms as a representation for functional programs. The programs are translated to categorical combinators, which are represented by sequential algorithms. The sequential algorithms they use are a somewhat simplified version of Berry and Curien's. The tree structure of algorithms is made explicit, and cell names become the concatenation of labels found on branches of a tree from the root to a leaf. Values are stored at the leaves. No provision is made for the *from* construct; in our vocabulary, only *valof* and *result* nodes exist. The operational semantics employed appears to be a version of CDS01.

In [50] a loop-detecting interpreter for a lazy, higher-order language is described. The standard approach to detecting loops is to check for a recursive function being called twice with the same arguments. But this does not work for higher-order functions. Using sequential algorithms one can get around this problem. The knowledge of what input cells a particular output cell depends on makes it possible to detect when a cell depends on itself. To do this, whenever we encounter a *valof* c while trying to compute the value of a cell, we keep track of c and all the cells c itself depends on. A cell is called *detectably bottom* when it either depends on itself, or it depends on a detectably bottom cell.

When having to answer the question $fix\ f\ ?\ c$, Hughes and Ferguson attempt to evaluate c in the chain of increasing approximations to the fixpoint of f: $\perp, f.\perp, f^2.\perp, \ldots$, either getting a value, or showing that c is detectably bottom. The key point is that, in a finite dcds, there is a bound on the number of unrollings of the fixpoint that must be made before it becomes clear that c cannot filled; this bound is simply the number of distinct cells in the dcds.

In a later paper [33], an abstract interpretation based on sequential algorithms is developed for a higher-order, lazy language. The implementation is reported as being orders of magnitude faster for higher-order programs than competing approaches, such as frontiers [45] and pending analysis [88]. The problem the implementation suffers from is space-inefficiency. This does not seem surprising, given that it uses a CDS01-like operational semantics. In more recent work, Hughes, Hunt, and Runciman [51] report on attempts at overcoming this problem.

Our own approach to abstract interpretation and loop-detection is somewhat different because we are using CDS02. We shall discuss this issue at length when we present our refinement type inference system in Chapter 6.

2.6 Refinement type inference for Standard ML

In the Freeman-Pfenning framework for refinement type inference [35, 36], only datatypes can be refined, and the refinements are specified using a *rectype* statement. For example, here is a polymorphic version of the refinement mentioned in Section 1.3, of empty, one element, and two or more element lists:

$$
\begin{array}{l}
\texttt{datatype}\ \alpha\ \mathit{list}\ \texttt{=}\ \texttt{nil | cons of}\ \alpha * \alpha\ \mathit{list} \\
\texttt{rectype}\ \alpha\ \mathit{empty}\ \texttt{=}\ \texttt{nil} \\
\quad \texttt{and}\ \alpha\ \mathit{singleton}\ \texttt{=}\ \texttt{cons}\ (\alpha, \texttt{nil}) \\
\quad \texttt{and}\ \alpha\ \mathit{long}\ \texttt{=}\ \texttt{cons}\ (\alpha, \texttt{cons}(\alpha, \alpha\ \top_{list})) \\
\quad \texttt{and}\ \alpha\ \bot_{list}\ \texttt{=}\ \texttt{bottom}\ (\mathit{list})
\end{array}
$$

This definition would result in the following refinement type lattice:

$$
\begin{tikzcd}
 & \alpha\ \top_{list} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\alpha\ \mathit{empty} \arrow[dr, dash] & \alpha\ \mathit{singleton} \arrow[d, dash] & \alpha\ \mathit{long} \arrow[dl, dash] \\
 & \alpha\ \bot_{list} &
\end{tikzcd}
$$

Note that the refinement types are kept separate from the regular types. In particular, the refinement type $\alpha\ \top_{list}$ gets automatically generated to correspond to the regular type $\alpha\ \mathit{list}$.

Given such a *rectype* declaration, the Freeman-Pfenning system automatically generates the following type for the constructor *cons*:

$$
\begin{array}{ll}
\texttt{cons} : & (\alpha * \alpha\ \mathit{empty}) \to \alpha\ \mathit{singleton}\ \wedge \\
 & (\alpha * \alpha\ \mathit{singleton}) \to \alpha\ \mathit{long}\ \wedge \\
 & (\alpha * \alpha\ \mathit{long}) \to \alpha\ \mathit{long}.
\end{array}
$$

In addition, their system can infer the following type for the polymorphic *map* function (written in a slightly different way than the example from Section 1.3):

$$
\begin{array}{ll}
\forall \alpha\beta.\ (\alpha \to \beta) \to & \alpha\ \mathit{empty}\ \to \beta\ \mathit{empty}\ \wedge \\
 & \alpha\ \mathit{singleton}\ \to \beta\ \mathit{singleton}\ \wedge \\
 & \alpha\ \mathit{long}\ \to \beta\ \mathit{long}.
\end{array}
$$

The ability to define refinements of parametrized types enables *rectype* declarations such as the following, which distinguishes even and odd length lists of booleans (*runit* stands for the empty tuple refinement type):

datatype $blist$ = nil | cons of $bool * list$
rectype bev = cons ($\top_{bool} * bod$) | nil ($runit$)
    and bod = cons ($\top_{bool} * bev$)

The refinement type inference algorithm works (roughly) by obtaining a regular type for an expression, then trying out all possible refinements of that type, and using refinement type inference rules to reach a result type. In the case of refinement type variables, all possible instantiations at a particular type must be considered. Pending analysis is used for fixpoints.

The main difficulties with this approach seem to be caused by instantiations of polymorphic refinement type variables. There are two problems: In many cases, one cannot use the polymorphic version of a function and get the best refinement type, and, when higher-order functions are used, the number of possible refinements gets very large, which leads to inefficiency.

As an example, suppose we have refined *bool* by *true* and *false*, and we want to derive a refinement type for the following program:

```
let val not = fn x => if x then false else true
    val double = fn f => fn x => f (f x)
in double not true
end;
```

The type of *double* is $\forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha$. When *double* is applied to *not*, since the regular type of *not* is $bool \to bool$, we need to instantiate α to all possible refinements of *bool*, which leads to the following refinement type for *double*:

$$
\begin{aligned}
&(true \to true) \to true \to true \ \wedge \\
&(false \to false) \to false \to false \ \wedge \\
&(\top_{bool} \to \top_{bool}) \to \top_{bool} \to \top_{bool} \ \wedge \\
&(\bot_{bool} \to \bot_{bool}) \to \bot_{bool} \to \bot_{bool}.
\end{aligned}
$$

Using this refinement type for *double*, since *not* has refinement type $true \to false \wedge false \to true$ (and, implicitly, $\top_{bool} \to \top_{bool}$), the best refinement type we can get for *double not* is $\top_{bool} \to \top_{bool}$, and hence the entire program has type $\top_{bool}$. The only way the more precise type of *true* can be obtained is if *double* is no longer a polymorphic function. If we specify that variable x in *double* is really a boolean, the refinement type of *double* becomes an intersection of 112 components, including pieces that enable us to infer $true \to true \ \wedge \ false \to false$ as the refinement type for *double not*. Using that type, we can get *true* as the type for the program.

2.7 Colson's work on intensional expressiveness

We could define the minimum of two integers in unary representation in a natural way by the following rewrite system:

$$
\begin{aligned}
&min(x, 0) = 0 \\
&min(0, x) = 0 \\
&min(S(x), S(y)) = S(min(x, y))
\end{aligned}
$$

Note that this is not a primitive recursive ($\mathcal{PR}$) algorithm; there is simultaneous recursion on two inputs.

We need to distinguish between the function min defined above, and an algorithm min_a for min. Intuitively, by applying the rewrite rules, the algorithm $min_a(n,p)$ computes its result in $O(min(n,p))$ time (it takes exactly $min(n,p)+1$ steps).

Colson studied the expressibility of minimum in the context of $\mathcal{PR}$ algorithms [19, 20]. He established that $\mathcal{PR}$ algorithms are inherently sequential: like sequential algorithms, they possess sequentiality indices. Moreover, $\mathcal{PR}$ algorithms are sequential in an even stronger sense. They suffer from "ultimate obstination" [20, 22]: at some point one argument must be chosen to be evaluated until the end. Using primarily the intensional semantics of lazy natural numbers ($LNAT$), which we exhibited earlier, he proved two main results:

Proposition 2.7.1 *There is no $\mathcal{PR}$ algorithm a of arity 2 satisfying:*

$$[\![a]\!](S^n(\bot), S^p(\bot)) = S^{min(n,p)}(\bot).$$

Proposition 2.7.2 *There is no $\mathcal{PR}$ algorithm which computes the minimum of two numbers n and p in unary representation, and is of complexity $O(min(n,p))$.*

However, there are many $\mathcal{PR}$ algorithms which compute the minimum of two integers. We define one below, using some auxiliary functions (see [57]):

$$pred(0) = 0$$
$$pred(S(x)) = x$$

$$sub(x, 0) = x$$
$$sub(x, S(y)) = pred(sub(x, y))$$

$$MIN(x, y) = sub(x, sub(x, y))$$

Note that in an operational interpretation of this definition, the algorithm $MIN_a(n,p)$ for MIN has a worst-case running time of $O(max(n,p))$. Let us call the elements of $LNAT$ of the form $S^k(0)$ *defined*, and the elements of the form $S^k(\bot)$ *partial*. The function MIN agrees with min on the defined elements of $LNAT$. They are different on the partial elements. By the $LNAT$ semantics we have:

$$\begin{aligned} [\![min]\!](S^n(\bot), S^p(\bot)) &= S^{min(n,p)}(\bot) \\ [\![MIN]\!](S^n(\bot), S^p(\bot)) &= \bot \end{aligned}$$

We can view Proposition 2.7.1 as an extensional result: $\mathcal{PR}$ algorithms can compute MIN but not min. Note that there are many other functions between min and MIN in the pointwise order. But it is the intensional aspect of Proposition 2.7.2 that is particularly interesting here: $\mathcal{PR}$ algorithms cannot compute minimum efficiently.

If we augment $\mathcal{PR}$ algorithms with functional arguments, we arrive at Gödel's system T [39]. In system T we can not only compute new functions (*e.g.,* the Ackermann function), but we can also compute minimum efficiently (system T can express an algorithm for min [19]). Thus, system T is more powerful than $\mathcal{PR}$ both extensionally and intensionally.

Colson's results are the first intensional expressiveness results for programming languages of which we are aware.

Chapter 3

Expressing Minimum

In this chapter, we begin our intensional explorations, by studying the expressibility of the minimum of two lazy natural numbers in CDS0. We expected to obtain results similar to Colson's in our study of sequential algorithms. After all, CDS0 is a sequential programming language by design: sequential algorithms compute sequential functions. It turns out, however, that sequential algorithms are sufficiently more powerful than $\mathcal{PR}$ algorithms to be able to compute minimum efficiently, but not powerful enough to compute the "natural" *min* function from Section 2.7. The parallel query construct of Brookes and Geva allows us to compute that function. This, of course, raises the question of whether the addition of query increases the intensional expressiveness of the language. We show that it does; in particular, the computation of various n-ary functions can be speeded up. However, this assumes non-parallel evaluation of CDS0.

Section 3.1 defines the dcds of lazy natural numbers and shows how it can be implemented in CDS0, along with various algorithms on the lazy natural numbers. In Section 3.2 we show that CDS0 cannot compute the *min* of Section 2.7, but can compute minimum efficiently. We exhibit an algorithm to do this. We introduce the extension of CDS0 with query, named CDSP, and define its semantics in Section 3.3. Section 3.4 shows the comparison of CDS0 and CDSP.

3.1 Implementing lazy natural numbers in CDS0

$LNAT$, the dcds of lazy natural numbers, is defined as follows: It has cells b_n, for $n \geq 0$, values 0 and 1, and the following accessibility relation: b_0 is initial, and $\{b_i = 1\} \vdash b_{i+1}$ (filling a cell with 1 enables the next cell). Intuitively, filling a cell with 1 means there might be more to follow, whereas 0 means we are done. $\langle D(LNAT), \subseteq \rangle$ is isomorphic to the domain $LNAT$ from the introduction. The encoding of the lazy natural numbers is:

$$\begin{aligned} S^n(\bot) &= \{b_i = 1 \mid i < n\}, \\ S^n(0) &= \{b_i = 1 \mid i < n\} \cup \{b_n = 0\}, \text{ for } n \geq 0, \\ S^\omega(\bot) &= \{b_i = 1 \mid i \geq 0\}. \end{aligned}$$

This mathematical definition of $LNAT$ can be implemented as CDS0 code in the following way:

```
letrec lnat = dcds
         cell B values 0,1
         graft (lnat.s) access B = 1
end;
```

We ask the interpreter to unroll the definition by displaying the first few cells and their access conditions:

```
# show more 3 lnat;
{
B values 0, 1,
(B.s) values 0, 1 access B=1,
((B.s).s) values 0, 1 access (B.s)=1}
```

Now let us define a few constants: $\perp$, 0, $S(\perp)$, 1, and $S^\omega(\perp)$:

```
let Bot   = {};
let Zero  = {B=0};
let S_bot = {B=1};
let One = {B=1,(B.s)=0};

let Srec = algo
      request B do
        output 1
      end
      request ((B.$V).s) do
        valof (B.$V) is
          1 : output 1
        end
      end
end;

let S_omega_bot = fix(Srec);
```

$S^\omega(\perp)$ is defined as the least fixpoint of the algorithm which in the base case fills B with 1, and recursively, if the previous cell contains 1, puts 1 into the current cell. All the algorithms we shall write on $LNAT$ will have a similar structure. Note how we have used a variable ($\$V$) to stand for a sequence of tags of $.s$ of any length (including 0).

Now we can write the successor algorithm; it is shown in Figure 3.1. It is only slightly more complicated than the algorithm for $S^\omega(\perp)$, but warrants further explanation because it is higher-order. Successor is defined as the fixpoint of a higher-order algorithm and it works as follows: If asked what B is, it immediately outputs 1 (the successor of anything is at least $S(\perp)$). In the general case, if asked what value an output cell holds, it asks what value the input cell immediately preceding it holds, and outputs the same value.

3.2 CDS0 and minimum

We begin by showing that sequential algorithms cannot compute min. The proof follows standard lines (*cf.* [6, 12]).

Proposition 3.2.1 *There is no sequential algorithm computing min.*

```
let succ_rec =
    algo
      request {}B do
        output output 1
      end
      request {}((B.$V).s) do
        output valof (B.$V)
      end
      request {(B.$V)=0}((B.$V).s) do
        output output 0
      end
      request {(B.$V)=1}((B.$V).s) do
        output output 1
      end
end;

let S = fix(succ_rec);
```

Figure 3.1: The successor algorithm

Proof: A sequential algorithm computes a sequential function. But *min* is not sequential, since it has no sequentiality index at $(\perp, \perp)$ for output cell b_0. In other words, there is no input cell which must be filled in order for *min* to fill b_0. (Actually, *min* has no sequentiality index at any $(S^n(\perp), S^n(\perp))$ for b_n, $n \geq 0$.) Therefore, no CDS0 algorithm can compute *min*. $\Box$

But this does not mean we cannot compute minimum efficiently in CDS0. Recall that the problem with $\mathcal{PR}$ algorithms was that they become "fixated" on one input. Sequential algorithms allow us to keep alternating between the two inputs, examining one cell at a time.

Proposition 3.2.2 *There is a sequential algorithm which computes the minimum of two numbers n and p in unary representation, and is of time complexity $O(min(n,p))$.*

Proof: The actual algorithm is listed in Appendix B.1. Even though it looks rather complicated it has the same basic structure as the previous $LNAT$ algorithms. For the purpose of the presentation, we shall assume the existence of a higher-level ML-like syntax for CDS0 algorithms, and discuss an algorithm written in that syntax. It is implicitly to be understood, however, that we are really referring to the CDS0 program.

In higher-level syntax, the algorithm looks like a simple sequential version of the *min* function definition from Section 2.7. We choose the left input to evaluate first.

algo left_min (n1, n2) =
  case n1 **of**
    0 ⇒ 0
    | S(x) ⇒ **case** n2 **of**
        0 ⇒ 0
        | S(y) ⇒ S(left_min(x, y))

The algorithm has the following property:

$$[\![\text{left\_min}]\!](S^n(0), S^p(0)) = S^{min(n,p)}(0),$$

so it does compute the minimum, and it works in time $O(min(n,p))$ by alternating between the inputs and examining one cell at a time. □

Note that the algorithm also satisfies:

$$[\![\text{left\_min}]\!](S^n(\perp), S^p(\perp)) = S^{min(n,p)}(\perp),$$

so Colson's Proposition 2.7.1 fails as well in the context of sequential algorithms.

The key difference between left_min and min_a is illustrated by their behavior on pairs of a totally defined and a partial element, such as $(S^n(\perp), S^n(0))$ (they agree on all other inputs):

$$\begin{aligned} [\![\text{left\_min}]\!](S^n(\perp), S^n(0)) &= S^n(\perp) \\ [\![min_a]\!](S^n(\perp), S^n(0)) &= S^n(0) \end{aligned}$$

This comparison makes it clear that min is a parallel function: it must evaluate its inputs in parallel in order to be able to determine when either one is defined. Also note that $[\![\text{left\_min}]\!]$ fits between min and MIN in the pointwise order.

We illustrate the behavior of `left_min` with the aid of the interpreter:

```
# left_min.(One,S_bot);
request? B;
--> 1
request? (B.s);
--> 0
request? ;
# left_min.(S_bot,One);
request? B;
--> 1
request? (B.s);
-->
request? ;
# left_min.(S_omega_bot,One);
request? B;
--> 1
request? (B.s);
--> 0
```

3.3 CDSP

We now consider the extension of CDS0 with the *query* construct, which we call CDSP, standing for "CDS Parallel." This was examined in detail by Brookes and Geva [12] from a denotational point of view. We are more interested in the operational aspect, since we want to know the running time of programs that use query. Consequently, we look at the changes necessary to the forest representation and semantics in order to accommodate query.

As when we first described the query construct in Section 2.4, we shall ignore issues related to ensuring that consistent inputs lead to the same output, and issues of the necessity of specifying future inputs in certain cases (such as for curried parallel algorithms). These issues are not related to our main concerns; we refer the reader to [12] for the extra notation required to handle such problems.

3.3.1 Forest semantics of query

We extend the set of forest instructions by queries, with the following general form:

$$\begin{array}{l} \mathit{Query}\ \{(c_1, i_1), \ldots, (c_n, i_n)\}\ \mathit{is} \\ \quad \{v_{11}, \ldots, v_{1n}\}\ :\ \mathit{ins}_1 \\ \quad \ldots \\ \quad \{v_{m1}, \ldots, v_{mn}\}\ :\ \mathit{ins}_m \\ \mathit{end} \end{array}$$

A query will have a number of patterns, each one of which is a vector of values extended with the special symbol "_". We introduce auxiliary notation to talk about patterns separately. In general, the ith pattern will have the form:

$$\begin{array}{l} \{(c_1, i_1), \ldots, (c_n, i_n)\}\ \mathit{is} \\ \quad \{v_{i1}, \ldots, v_{in}\}. \end{array}$$

Evaluating a pattern involves evaluating all the cells for which the corresponding pattern position is not "_" in parallel, and verifying that the values match. There are three possible answers:

1. The values match, in which case we return the special value *match*.
2. There is at least one value that does not match. We return *no match*.
3. We do not have enough information to decide if we have a match. In this case we issue a *residual pattern* which asks for the values of just those cells whose values we still need to know.

This is summarized in the following evaluation rule for patterns, using the conventions that for any v, "_" $\sqsubseteq v$, and $(v_1, \ldots, v_n) \sqsubseteq (v'_1, \ldots, v'_n)$ when, for each i, $v_i \sqsubseteq v'_i$.

$$(\textsc{Pat})\quad \frac{\left.\begin{array}{l} x_{i_1}\ ?\ c_1 \ \rightarrow\ v'_1, \\ \ldots \\ x_{i_n}\ ?\ c_n \ \rightarrow\ v'_n \end{array}\right\}\ \text{and}\ \left\{\begin{array}{l} (v'_1, \ldots, v'_n) \sqsupseteq (v_1, \ldots, v_n) \\ (v'_1, \ldots, v'_n)\ \text{incomparable to}\ (v_1, \ldots, v_n) \\ (v'_1, \ldots, v'_n) \sqsubseteq (v_1, \ldots, v_n) \end{array}\right.}{\left.\begin{array}{l} \{(c_1, i_1), \ldots, (c_n, i_n)\}\ \mathit{is} \\ \quad \{v_1, \ldots, v_n\}. \end{array}\right\}\ ?\ x_1 \cdots x_n c' \ \rightarrow\ \left\{\begin{array}{l} \text{match} \\ \text{no match} \\ \text{residual pattern} \end{array}\right.}$$

When executing a query, we will evaluate all the patterns in parallel. There are also three possibilities:

1. At least one pattern matches (it is fine if several patterns match, since we assume outputs are the same in that case), in which case we execute the appropriate instruction.
2. No pattern matches, in which case we return Ω.
3. Evaluation of all patterns results in residual patterns. In that case we return a *residual query* by putting together the residual patterns. We will not provide details on constructing such queries.

We present below the rule for evaluation of query instructions. $Pattern_k$ stands for the kth pattern in the query, and $1 \leq k \leq m$.

$$
(\textsc{Qry}) \quad \frac{\left\{\begin{array}{l} Pattern_k \ ?\ x_1 \cdots x_n c' \ \rightarrow \ \text{match} \\ \forall k.\ Pattern_k \ ?\ x_1 \cdots x_n c' \ \rightarrow \ \text{no match} \\ \forall k.\ Pattern_k \ ?\ x_1 \cdots x_n c' \ \rightarrow \ \text{residual pattern} \end{array}\right.}{\left.\begin{array}{l} Query\ \{(c_1, i_1), \ldots, (c_n, i_n)\}\ is \\ \quad \{v_{11}, \ldots, v_{1n}\} \ :\ ins_1 \\ \quad \ldots \\ \quad \{v_{m1}, \ldots, v_{mn}\} \ :\ ins_m \\ end \end{array}\right\} \ ?\ x_1 \cdots x_n c' \ \rightarrow \ \left\{\begin{array}{l} ins_k \ ?\ x_1 \cdots x_n c' \\ \Omega \\ \text{residual query} \end{array}\right.}
$$

3.3.2 CDSP and minimum

The addition of the parallel query construct enables us to compute *min*, which is essentially a generalization of parallel-or to integer arguments. The program is shown in Appendix B.2. Note that it is actually simpler than the CDS0 program for `left_min`. Again, for the purpose of the presentation, we use a higher-level syntax. In that syntax, the program looks almost the same as the definition of the *min* function from the introduction:

```
algo min (n1, n2) =
     query (n1, n2) is
          (0, _) ⇒ 0
        | (_, 0) ⇒ 0
        | (S(x), S(y)) ⇒ S(min(x, y))
```

The program is clearly efficient, examining two cells at a time. We then obtain the following:

Proposition 3.3.1 *There is a CDSP program computing min.*

3.4 CDS0 versus CDSP

We have seen that both CDS0 and CDSP can compute the minimum of two lazy natural numbers efficiently. This raises the question of whether the addition of deterministic parallelism to CDS0 buys us any intensional power. There actually appears to be a folk conjecture that deterministic parallelism is not "useful." The claim is that even though deterministic parallel features may increase the extensional expressiveness of a language, they are expensive to use and the additional expressiveness is not useful in practice, because it applies only to computations that are unbounded. In our terms, the claim is that deterministic parallelism may increase extensional, but not intensional expressiveness.

This conjecture is false. Deterministic parallelism does add intensional expressiveness. The deterministic query construct of CDSP is sufficiently general to allow a speedup in the computation of many different functions. When computing certain n-ary functions, the query construct allows us to construct a tree of processes of logarithmic depth. We illustrate with n-ary disjunction.

For notational simplicity, we define a separate function for each value of n, and we assume n is a (fixed) power of 2. We have already defined *por* for two arguments. Here is the general case:

algo $\mathrm{por_n}\ (\mathrm{b_1}, \ldots, \mathrm{b_n}) =$
$\quad \mathrm{por}\ (\mathrm{por_{n/2}}\ (\mathrm{b_1}, \ldots, \mathrm{b_{n/2}}),$
$\quad\quad \mathrm{por_{n/2}}\ (\mathrm{b_{n/2+1}}, \ldots, \mathrm{b_n}))$

When computing por_n, in order to fill the output cell we query in parallel two cells. In order to fill those cells, we query two more for each. Intuitively, after a depth of $log\ n$ queries we reach our n inputs. Therefore, we compute the result in time $O(log\ n)$. In CDS0, since we must examine the inputs sequentially, we can only compute the result in time $O(n)$.

We can formalize this by instrumenting our operational semantics to keep track of depth of the computation. We only do this for result, query, valof, application, and product, as the others are similar.

The new rules will have the form $forest, t\ ?\ c\ \rightarrow\ v, t'$, where t stands for the time (or depth) at which the question is asked, and t' for the time at which an answer is issued. The modifications for result and valof are simple.

$$(\text{RESULT'}) \qquad Result\ v', t\ ?\ x_1 \cdots x_n c'\ \rightarrow\ v', t+1$$

$$(\text{VALOF'}) \quad \frac{x_p, t\ ?\ c\ \rightarrow \begin{cases} v_i, t' \\ v, t',\ \text{and}\ \forall i.\ v_i \neq v \\ \Omega, t' \end{cases}}{\left.\begin{array}{l} Valof\ (c,i)\ is \\ \quad v_1\ :\ ins_1 \\ \quad \ldots \\ end \end{array}\right\}, t\ ?\ x_1 \cdots x_n c'\ \rightarrow \begin{cases} ins_1, t'+1\ ?\ x_1 \cdots x_n c' \\ \Omega, t'+1 \\ output^{p-1}\ valof\ c, t'+1 \end{cases}}$$

For query, the difference is that we are evaluating the patterns in parallel, and, within each pattern, the cells are also evaluated in parallel. The depth of a pattern evaluation will depend on the *maximum* of the depths of its sub-computations. The depth of a successful query will be the minimum of the depths of the matching pattern evaluations; an unsuccessful query will have a depth that is the maximum of the depths of all pattern evaluations.

$$(\text{PAT'}) \quad \frac{\left.\begin{array}{l} x_{i_1}, t\ ?\ c_1\ \rightarrow\ v'_1, t_1 \\ \ldots \\ x_{i_n}, t\ ?\ c_n\ \rightarrow\ v'_n, t_n \end{array}\right\}\ \text{and} \begin{cases} (v'_1, \ldots, v'_n) \sqsupseteq (v_1, \ldots, v_n) \\ (v'_1, \ldots, v'_n)\ \not\sqsupseteq, \not\sqsubseteq\ (v_1, \ldots, v_n) \\ (v'_1, \ldots, v'_n) \sqsubseteq (v_1, \ldots, v_n) \end{cases}}{\left.\begin{array}{l} \{(c_1, i_1), \ldots, (c_n, i_n)\}\ is \\ \quad \{v_1, \ldots, v_n\}. \end{array}\right\}, t\ ?\ x_1 \cdots x_n c'\ \rightarrow \begin{cases} \text{match}, T \\ \text{no match}, T \\ \text{residual pattern}, T \end{cases}}$$

$$(\text{QRY'}) \quad \frac{\begin{cases} Pattern_k, t\ ?\ x_1 \cdots x_n c'\ \rightarrow\ \text{match}, t_k \\ \forall k.\ Pattern_k, t\ ?\ x_1 \cdots x_n c'\ \rightarrow\ \text{no match}, t_k \\ \forall k.\ Pattern_k, t\ ?\ x_1 \cdots x_n c'\ \rightarrow\ \text{residual pattern}, t_k \end{cases}}{\left.\begin{array}{l} Query\ \{(c_1, i_1), \ldots\}\ is \\ \quad \{v_{11}, \ldots, v_{1n}\}\ :\ ins_1 \\ \quad \ldots \\ \quad \{v_{m1}, \ldots, v_{mn}\}\ :\ ins_m \\ end \end{array}\right\}, t\ ?\ x_1 \cdots x_n c'\ \rightarrow \begin{cases} ins_k, T_{min}\ ?\ x_1 \cdots x_n c' \\ \Omega, T_{max} \\ \text{residual query}, T_{max} \end{cases}}$$

where $T = 1 + \max\{t_1, \ldots, t_n\}$, $T_{max} = \max\{t_1, \ldots, t_n\}$, and $T_{min} = \min\{t_{k_1}, \ldots, t_{k_m}\}$, for all k_i such that $Pattern_{k_i}, t\ ?\ x_1 \cdots x_n c'\ \rightarrow\ \text{match}, t_{k_i}$.

Finally, emblematic of the modified CDS02 rules is application. We also show the new rule for product, which is needed in what follows.

$$(\textsc{App'})\qquad \frac{A, t\ ?\ Bc' \to \left\{\begin{array}{c}\Omega\\ \mathit{valof}\ c\\ \mathit{output}\ v'\end{array}\right\}, t'}{A.B, t\ ?\ c' \to \left\{\begin{array}{c}\Omega\\ \Omega\\ v'\end{array}\right\}, t'+1}$$

$$(\textsc{Prod'})\qquad \prod_{i=1}^{n} A_i, t\ ?\ (c.i) \to A_i, t+1\ ?\ c$$

Example 3.4.1 *Suppose we just want to ask a question of a ground state. Let us consider the state* $\{B = tt\}$. *Its forest representation is* $\mathrm{Tree}(B, \mathrm{Result}\ tt)$, *so we have:*

$$\{B = tt\}, 0\ ?\ B\ \to\ tt, 1.$$

Example 3.4.2 *We have already seen the internal representation of land in Figure 2.5. We have:*

$$land.\{(B.1) = tt, (B.2) = tt\}, 0\ ?\ B \to tt, 4,$$

because there are two valof and two result instructions along the way.

We are now ready to prove that n-ary disjunction works in logarithmic time.

Proposition 3.4.3 $por_n(b_1, \ldots, b_n), 0\ ?\ B\ \to v, t$, *where* $t \leq 4 \log n$.

Proof: By induction on n, which is always a power of 2.

In the base case we have, $por(b_1, b_2), 0\ ?\ B\ \to\ v, 4$, since we execute App', Query', Prod', and Result'. But $4 = 4\log 2$, so the proposition holds for $n = 2$. Suppose it holds for n. Then

$$por_{2n}(por_n(b_1, \ldots, b_n), por_n(b_{n+1}, \ldots, b_{2n})), 0\ ?\ B\ \to\ v, t,$$

where $t = 1 + 1 + \max\{1 + 4\log n, 1 + 4\log n\} = 3 + 4\log n$. But $4 \log 2n = 4 + 4\log n > t$. $\square$

So we have established that:

Proposition 3.4.4 *CDSP is intensionally more expressive than CDS0.*

3.5 Discussion

The sequentiality of the primitive recursive algorithms is manifested by their ability to recur on only one input. This makes them "ultimately obstinate," and they are not able to express an efficient algorithm for minimum.

The sequentiality of Berry-Curien algorithms is "by design." A sequential algorithm computes a sequential function, by only choosing one sequentiality index at a time, even if more than one exists. However, sequential algorithms are more expressive than primitive recursive algorithms: there is a sequential algorithm that computes a version of the minimum function efficiently, but not the "natural," inherently parallel, minimum function.

The addition of deterministic parallelism to CDS0 allowed us to compute the "natural" version of the minimum function, but CDS0 was already able to express an efficient minimum algorithm. However, the addition of deterministic parallelism did add intensional expressiveness, contradicting a folk conjecture. The computation of a number of functions can be speeded up, such as n-ary disjunction.

Note that there is a certain sense, however, in which our comparison of CDS0 and CDSP is not "fair." It is possible to imagine parallel evaluation strategies for CDS0 (*cf.* Curien [26]). Such parallel evaluation would not work well with CDS02, but in CDS01, with its tables, we could have *eager* computation which fills the table without waiting for a question. This could, of course, lead to a lot of useless computation, so it may be possible that we get good time-efficiency, but poor work-efficiency. We return to this point in the concluding chapter.

Chapter 4

Circuit Semantics

This chapter consists of two parts, both concerned with establishing relative intensional expressiveness results for parallel extensions of PCF, and both utilizing circuit semantics as the main tool. Circuit semantics associates a gate with each basic construct of the language, and takes the meaning of a program to be a circuit. The dimensions of the circuit enable reasoning about running time and work required for execution. In the first part of the chapter, we compare four deterministic extensions of PCF: parallel-or, parallel conditionals on booleans and integers, and deterministic query [12]. To aid us in this comparison we introduce a naïve version of the circuit semantics (first reported in [13]), which enables us to talk about relative *depth* of an implementation. This notion is good enough to produce a hierarchy of intensional expressiveness: query is the most powerful, followed by parallel conditional on integers, while parallel-or and parallel conditional on booleans are equivalent and the weakest.

In the second part of the chapter, we compare deterministic query with a nondeterministic version (first presented in [27]). We refine the circuit semantics to allow us to talk about parallel time and parallel work required for execution, and we establish connections between the size and depth of a circuit representing a parallel PCF program to the time and work required to execute it under call-by-speculation [49], parallel call-by-value, and parallel eager evaluation. We also relate the circuit dimensions of a program to the time and number of processors required to execute it in the PRAM model.

In order to be able to compare the two versions of query, we are forced to make a hardware assumption which is equivalent to having the ability to detect undefined inputs. This makes a subset of the programs using nondeterministic query return a deterministic result. The assumption is reasonable from a practical point of view and has been used in various studies of consensus problems in distributed systems [34]. The effect of this assumption is to render our problem similar to one from computational complexity, that of comparing monotone and De Morgan boolean circuits. It turns out that parallel PCF programs are intensionally equivalent to boolean circuits for a certain class of functions involving undefined inputs. This connection allows us to use strong results from complexity theory to establish intensional expressiveness results.

Section 4.1 describes the slightly different version of PCF we are using, and the deterministic parallel extensions we shall be comparing. A first version of the circuit semantics is introduced in Section 4.2 and is used to obtain the first intensional separation results in Section 4.3. Section 4.4 describes the recursion-free version of PCF we use for the nondeterministic extension, and introduces the nondeterministic query. The circuit semantics is refined in Section 4.5 and the method for making the comparison between determinism and nondeterminism is outlined in Section 4.6. Section 4.7 shows a connection between our question and circuit complexity, which is used in Section 4.8 to obtain the separation of deterministic and nondeterministic query. Finally, Section 4.9 provides a discussion of the results.

4.1 PCF and deterministic parallel extensions

4.1.1 PCF

In addition to the standard PCF constants listed in Figure 2.1, we assume the existence of a constant-time equality test for integers:

$$= \,:\, \iota \to \iota \to o$$

with the obvious operational semantics. Traditionally, the equality test is implemented using recursion (*cf.* [81]), but this would render some of the issues of interest to us moot. The reason for this is that in what follows we will want to know when one construct can be implemented in terms of others without using recursion. Since we are not using integers in unary representation, it would be unreasonable to have to use recursion to check for equality. In fact, a more realistic logarithmic-time test would not invalidate our results; we chose a constant-time test because it is simpler.

Let $FV(M)$ stand for the set of free variables of term M. If $FV(M) = \emptyset$ then the term M is *closed*, else it is *open*. The closed terms of ground type are referred to as *programs*.

4.1.2 Parallel-or and parallel conditionals

The parallel extensions of PCF studied in Plotkin's seminal paper [70] are: *por*, pif_o (parallel conditional on booleans), and pif_ι (parallel conditional on integers). The extension of PCF with any of these functions is fully abstract with respect to the standard denotational semantics. The parallel conditionals are defined as follows:

$$\begin{aligned}
&\mathit{pif}_\sigma \,:\, o \to \sigma \to \sigma \to \sigma\\
&\mathit{pif}_\sigma \;\bot\; x\; x = x\\
&\mathit{pif}_\sigma \;\mathit{tt}\; x\; \bot = x\\
&\mathit{pif}_\sigma \;\mathit{ff}\; \bot\; x = x
\end{aligned}$$

for $\sigma = \iota, o$. *Por*, pif_o, and pif_ι are known to be extensionally equivalent [26, 81], *i.e.,* one can be implemented in terms of another. The question we address is whether they are intensionally equivalent. Interestingly, the answer turns out to be negative.

4.1.3 Query

Another parallel deterministic extension we are interested in exploring is *query*. We have already encountered query in the context of concrete data structures, but the construct is quite general, and in this chapter we use it to extend PCF. Figure 4.1(a) shows the PCF-like syntax we envision for query in yet another example of a program for parallel-or.

The general form of the query syntax is shown in Figure 4.1(b). The $x_i^{\sigma_i}$ ($\sigma_i \in \{o, \iota\}$) are variables, the p_i are patterns, and the M_i^τ are PCF terms. A pattern is a vector of length n (the number of variables in the query), with each element being either a variable y, a closed term e of ground type, or the "don't care" symbol "_". All of the variables in one pattern must be distinct.

We distinguish between two versions of the query construct: deterministic and nondeterministic. In the first case, we shall require the same output for all *consistent* inputs. Let $p_1 = (v_1, \ldots, v_n)$

$$\begin{array}{rl} por \equiv \lambda xy.\ \text{query}\ (x,y)\ \text{is} & \\ (tt,\ \_) \Rightarrow tt & \\ \mid (\_,\ tt) \Rightarrow tt & \\ \mid (ff,\ ff) \Rightarrow ff & \end{array} \qquad \begin{array}{l} \text{query} : (\sigma_1 \times \cdots \times \sigma_n) \rightarrow \tau \\ \text{query}\ (x_1^{\sigma_1}, \ldots, x_n^{\sigma_n})\ \text{is} \\ \quad p_1 \Rightarrow M_1^\tau \\ \quad \ldots \\ \quad p_k \Rightarrow M_k^\tau \end{array}$$

Figure 4.1: (a) parallel-or, (b) query syntax

and $p_2 = (w_1, \ldots, w_n)$ be two patterns. We call the two patterns *consistent* (written $p_1 \Uparrow p_2$) if $\forall i$, either $\mathcal{D}[\![v_i]\!] \sqsubseteq \mathcal{D}[\![w_i]\!]$ or $\mathcal{D}[\![v_i]\!] \sqsupseteq \mathcal{D}[\![w_i]\!]$. Since the elements of patterns come from flat domains, this formulation of consistency coincides with the conventional notion of "having an upper bound." We extend the standard semantics with $\mathcal{D}[\![\_]\!]\rho = \bot$, and note that a variable in a pattern will always be consistent with anything, since it is the equivalent of a "don't care."

Example 4.1.1 *We present some examples of consistent and inconsistent patterns. We have*

$$(\_, 1) \Uparrow (0, \_),\ \ (\_, y) \Uparrow (x, \_),\ \ (x, 1) \Uparrow (0, y),\ \ \textit{and also}\ (x, 1) \Uparrow (0, x).$$

On the other hand,

$$(0, 1) \not\Uparrow (0, 0)\ \ \textit{and}\ (\_, 1) \not\Uparrow (0, 0).$$

A deterministic query has the property that it produces the same output for all consistent inputs, *i.e.,* given two patterns p_i, p_j, if $p_i \Uparrow p_j$ then $\mathcal{D}[\![M_i]\!] = \mathcal{D}[\![M_j]\!]$. The determinism restriction makes it fairly easy to define a semantics, shown in Figure 4.2. We use Q to refer to the general form of query, from Figure 4.1(b). *Amb* is McCarthy's ambiguity operator [62]:

$$amb(\bot, x) = x, \quad amb(x, \bot) = x, \quad amb(x, y) = x\ \textbf{or}\ y,$$

which behaves essentially like a parallel-or when only one argument is defined, and performs an arbitrary choice between the arguments if both are defined. Amb_k is k-ary *amb*. Because of the determinism constraints, it will always be the case that *amb* will behave deterministically, *i.e.,* if we have $amb(x, y)$ with both x, y defined, then $x = y$. Consequently, the meaning of a query will be a continuous function.

We use the notation $\vec{x}$ for $(x_1^{\sigma_1}, \ldots, x_n^{\sigma_n})$, and $\mid$ for concatenating environments, with the simple properties: $\rho \mid \bot = \bot \mid \rho = \rho$. It is not possible to have multiple bindings for the same variable, because of our requirement that all variables in one pattern should be distinct. $\wedge$ is parallel-and, with the properties: $tt\ \wedge\ tt = tt$, $ff\ \wedge\ \bot = ff$, $\bot\ \wedge\ ff = ff$.

The auxiliary semantic function $\mathcal{D}_{pat}$ defines the meaning of a pattern match. It keeps track of whether the pattern match succeeds and of any bindings generated in the process. A pattern match succeeds when each element of the pattern matches its corresponding input. Since we can have variables in the patterns, we may generate bindings. The results of element-wise matching comparisons are combined with parallel-and, and the environment extended with any newly generated bindings.

We work out an example in detail in order to illustrate the semantics. Consider the following query:

$\mathcal{D}_{pat}$: Patterns $\rightarrow$ Environments $\rightarrow (D_{bool} \times$ Environments)

$$
\begin{aligned}
&\mathcal{D}[\![Q]\!]\rho = \mathrm{amb}_k(\mathcal{D}[\![\vec{x} \text{ is } p_1 \Rightarrow M_1^\tau]\!]\rho, \ldots, \mathcal{D}[\![\vec{x} \text{ is } p_k \Rightarrow M_k^\tau]\!]\rho) \\
&\mathcal{D}[\![\vec{x} \text{ is } p \Rightarrow M^\tau]\!]\rho = \begin{cases} \mathcal{D}[\![M^\tau]\!](\rho \mid \mu), & \text{if } \mathcal{D}_{pat}[\![\vec{x} \text{ is } p]\!]\rho = (tt, \mu) \\ \perp, & \text{if } \mathcal{D}_{pat}[\![\vec{x} \text{ is } p]\!]\rho = (\mathit{ff}, \mu) \end{cases} \\
&\mathcal{D}_{pat}[\![(x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}) \text{ is } (e_1^{\sigma_1}, \ldots, e_n^{\sigma_n})]\!]\rho = (b_1 \wedge \cdots \wedge b_n, \mu_1 \mid \cdots \mid \mu_n), \\
&\qquad\qquad \text{where } (b_i, \mu_i) = \mathcal{D}_{pat}[\![x_i^{\sigma_i} \text{ is } e_i^{\sigma_i}]\!]\rho \\
&\mathcal{D}_{pat}[\![x^\sigma \text{ is } e^\sigma]\!]\rho = \begin{cases} (\mathcal{D}[\![x^\sigma]\!]\rho = \mathcal{D}[\![e^\sigma]\!]\rho, \perp), & \text{if } e^\sigma \text{ is closed} \\ (tt, y^\sigma \mapsto x^\sigma), & \text{if } e^\sigma \equiv y^\sigma \\ (tt, \perp), & \text{if } e^\sigma \equiv \_ \end{cases}
\end{aligned}
$$

Figure 4.2: Denotational semantics for deterministic query

$$
\begin{aligned}
Q \equiv \text{query } &(x_1, x_2) \text{ is} \\
&\quad (x, 1) \Rightarrow M_1 \\
&\mid (0, y) \Rightarrow M_2.
\end{aligned}
$$

According to the semantics, we have:

$$\mathcal{D}[\![Q]\!]\rho = \mathrm{amb}(\mathcal{D}[\![x_1 x_2 \text{ is } (x,1) \Rightarrow M_1]\!]\rho, \mathcal{D}[\![x_1 x_2 \text{ is } (0,y) \Rightarrow M_2]\!]\rho).$$

The arguments to the ambiguity operator are:

$$\mathcal{D}[\![x_1 x_2 \text{ is } (x,1) \Rightarrow M_1]\!]\rho = \begin{cases} \mathcal{D}[\![M_1]\!](\rho \mid x \mapsto x_1), & \text{if } \rho(x_2) = 1 \\ \perp, & \text{otherwise.} \end{cases}$$

$$\mathcal{D}[\![x_1 x_2 \text{ is } (0,y) \Rightarrow M_2]\!]\rho = \begin{cases} \mathcal{D}[\![M_2]\!](\rho \mid y \mapsto x_2), & \text{if } \rho(x_1) = 0 \\ \perp, & \text{otherwise.} \end{cases}$$

One can easily see why having variables in the patterns is equivalent to a "don't care"; the only difference is that the environment ρ gets extended with a new binding. The above equations were obtained with the aid of the $\mathcal{D}_{pat}$ semantics:

$$
\begin{aligned}
\mathcal{D}_{pat}[\![x_1 x_2 \text{ is } (x,1)]\!]\rho = (\rho(x_2) = 1, x \mapsto x_1), &\text{ because} \\
\mathcal{D}_{pat}[\![x_1 \text{ is } x]\!]\rho &= (tt, x \mapsto x_1) \\
\mathcal{D}_{pat}[\![x_2 \text{ is } 1]\!]\rho &= (\rho(x_2) = 1, \perp) \\
\mathcal{D}_{pat}[\![x_1 x_2 \text{ is } (0,y)]\!]\rho = (\rho(x_1) = 0, y \mapsto x_2), &\text{ because} \\
\mathcal{D}_{pat}[\![x_1 \text{ is } 0]\!]\rho &= (\rho(x_1) = 1, \perp) \\
\mathcal{D}_{pat}[\![x_2 \text{ is } y]\!]\rho &= (tt, y \mapsto x_2)
\end{aligned}
$$

It should be noted that since the two patterns are consistent (*cf.* Example 4.1.1), we must have $\mathcal{D}[\![M_1]\!] = \mathcal{D}[\![M_2]\!]$ if our query is to be deterministic.

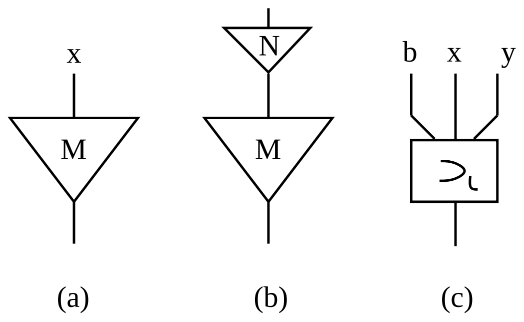

Figure 4.3: (a) $\lambda x.\ M$, (b) $(\lambda x.\ M)N$, (c) $\supset_\iota b\ x\ y$

4.2 Circuit semantics: first approach

In order to compare *por*, pif_o, and pif_ι, we find it useful to view PCF programs as circuits. There are several reasons for this. First, it enables us to reason based on the last gate used in the circuit. Viewing a program as a circuit reduces the number of cases we need to consider. Second, the running time of the program *loosely* corresponds to the depth of the circuit. At this stage, we are only interested in the depth of programs, *i.e.,* closed terms of ground type, so we need not worry about complications caused by higher-order terms. Also, a *loose* correspondence is fine, since we only need to distinguish programs that use recursion from programs which do not. And third, circuits provide a visual and intuitive semantics. This is more than a cosmetic point: viewing programs as circuits enables us to find the connection with boolean circuits in the second part of this thesis.

The translation from PCF to circuits is simple. Figure 4.3 shows circuits for function definition, application, and a constant. A function denotes a circuit some of whose inputs are labelled with variables. Application substitutes a value for a variable, or, if we have a whole circuit, connects its output to the respective variable-labelled input. Note that higher-order functions can be treated in this framework as well, by using gates labelled with the function variable inside the circuit (for an example, see Figure 4.8 in Section 6). There are gates for the various constants. The only interesting case is the Y combinator. It gives rise to a special kind of circuit, a *dynamic circuit*, which can have subparts expanded dynamically as required during computation.

The semantics of circuits is based on PCF's operational semantics. Execution is demand-driven and begins at the output. The last gate in the circuit is activated. This gate may start evaluating one (or more, if it is parallel) of its inputs, leading to activity at further gates, and so on. If the computation terminates, the result will filter down to the output of the last gate.

Definition 4.2.1 *A circuit is static if it is the translation of a non-recursive PCF program.*

Definition 4.2.2 *A circuit is dynamic if it is the translation of a recursive PCF program.*

A circuit could have several inputs, but it always has just one output, so it is shaped as a tree.

Definition 4.2.3 *The depth of a static circuit is equal to the height of the underlying tree.*

Definition 4.2.4 *A circuit is constant-depth if it is either static, or a dynamic circuit which does not expand more than a fixed constant number of times (independent of the inputs).*

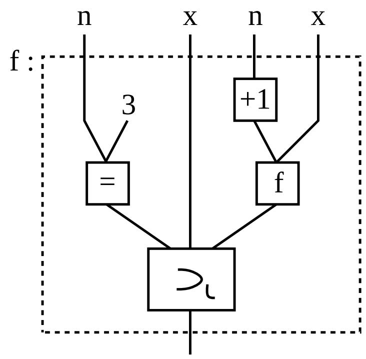

Figure 4.4: YF

Example 4.2.5 *To give an example of dynamic circuits, and to illustrate the difference between constant-depth and non-constant-depth dynamic circuits, consider the following PCF term:*

$$F = \lambda fnx.\ \supset_\iota (= n\ 3)\ x\ (f\ (+1\ n)\ x).$$

Figure 4.4 shows the circuit denoted by the recursive PCF term YF. We enclose a dynamic circuit in a box with dotted lines, to represent the fact that it can be expanded. The box is labelled with the name of the recursive part. The result of expanding the circuit once is shown in Figure 4.5.

The program $YF\,n$ for $0 \leq n \leq 3$ gives rise to a constant-depth dynamic circuit, while for $n > 3$ it results in a non-constant-depth dynamic circuit.

In the following, we are particularly interested in the constant-depth circuits. If two functions can be implemented in terms of each other with constant-depth circuits, we say that the two functions are *intensionally equivalent*.

4.3 Intensional separation for deterministic extensions

4.3.1 pif_ι versus por and pif_o

We begin by reviewing known implementations of the various functions.

Proposition 4.3.1 *por and pif_o are intensionally equivalent.*

Proof: We need constant-depth implementations of one in terms of the other. This can be done as follows (*cf.* [81]):

$$\begin{aligned} por &= \lambda xy.\ pif_o\ x\ tt\ y, \\ pif_o &= \lambda bxy.\ por\ (pand\ b\ x) \\ &\qquad\qquad\quad (pand\ (not\ b)\ y) \\ &\qquad\qquad\quad (pand\ x\ y), \end{aligned}$$

where *pand* is the parallel conjunction defined by:

$$pand = \lambda xy.\ not\ (por\ (not\ x)\ (not\ y)),$$

and we have generalized *por* to three arguments in the obvious way. □

It is known that pif_ι can implement pif_o (*cf.* [81]):

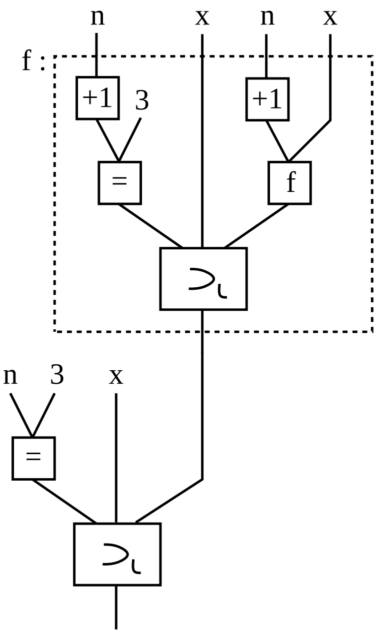

Figure 4.5: YF expanded once

$$pif_o = \lambda bxy.\ (= 1\ (pif_\iota\ b\ (\supset_\iota x\ 1\ 0)\ (\supset_\iota y\ 1\ 0))).$$

This implementation is also efficient. In view of the previous proposition, it follows that pif_ι can also implement *por* efficiently. However, the converse is false. The problem is that *por* can only start parallel subcomputations on booleans, whereas pif_ι operates in parallel on integers. The standard way of encoding pif_ι with *por* uses recursion (*cf.* [81]):

$$\begin{aligned}
&pif_\iota = YF\,0,\ \text{where}\\
&F = \lambda fnbxy.\ \supset_\iota (por\ (pand\ (= x\ n)\ (= y\ n))\\
&\qquad\qquad\qquad\quad (pand\ b\ (= x\ n))\\
&\qquad\qquad\qquad\quad (pand\ (not\ b)\ (= y\ n)))\\
&\qquad\qquad\quad n\\
&\qquad\qquad\quad (f\ (+1\ n)\ b\ x\ y).
\end{aligned}$$

This is clearly inefficient, because of the way the recursion unwinds, checking if x and y are equal to 0 first, then 1, and so on. But we cannot do any better. To show that, we prove first two lemmas which restrict the shape of any program computing pif_ι.

The point of the first lemma is that it is impossible to design boolean circuitry B which chooses between x and y and obeys all the requirements of pif_ι.

Lemma 4.3.2 *It is not possible to write a program in PCF + por that computes $pif_\iota\ b\ x\ y$ and is of the form $\supset_\iota B\ x\ y$, where B is a static circuit yielding a boolean.*

Proof: Without loss of generality, the issue is whether it is possible to write a PCF + *por* function B with the following properties:

1. If b is tt then B is tt,

b	$(= x\ y)$	B
tt	$\perp$	tt
ff	$\perp$	ff
$\perp$	tt	tt

Table 4.1: Requirements for function B

2. If b is ff then B is ff,

3. If $(= x\ y)$ is tt then B is tt.

Figure 4.1 shows some of the inputs and corresponding outputs for function B. For simplicity, we assume only b and $(= x\ y)$ are used in evaluating B. The same argument can be carried through with additional inputs, since b and $(= x\ y)$ *must* be used in evaluating B.

The last line of Table 4.1 implies by monotonicity that $B\ \mathit{ff}\ tt = tt$. But this violates the monotonicity condition raised by the second line. Therefore, no program of this form computes $\mathit{pif}_\iota\ b\ x\ y$. $\Box$

Our second lemma generalizes the first one.

Lemma 4.3.3 *It is not possible to write a program in PCF + por that computes $\mathit{pif}_\iota\ b\ x\ y$ and is of the form $\supset_\iota B\ N_1\ N_2$, where B, N_1, N_2 are static circuits yielding a boolean and two integers respectively.*

Proof: Intuitively, there are two possibilities for B: either it "chooses" between N_1 and N_2, or it is "hardwired" to always pick one of them. More precisely, we have two cases for the function computed by B:

1. B is non-constant. Since the program computes $\mathit{pif}_\iota\ b\ x\ y$, the result must be either x or y. There are an infinite number of possible inputs and outputs and N_1, N_2 are static circuits, so it is not possible to hard-code the output. B will sometimes return tt and sometimes ff. There are then three choices for what N_1, N_2 evaluate to:

   (a) They evaluate to x, y, respectively. But this is impossible by Lemma 4.3.2.

   (b) They both evaluate to $\mathit{pif}_\iota\ b\ x\ y$. The $\supset_\iota$ gate then does no work. Since N_1, N_2 both compute something of type integer, there are essentially two cases for the last gate used in their construction: (i) $\supset_\iota$ or (ii) $+1$ (-1 is handled similarly). In case (i) apply the same reasoning of this lemma. There cannot be an infinite sequence of $\supset_\iota$ gates which do nothing, since the circuit is static. It is not possible for all $\supset_\iota$ gates to do nothing since the output would then have to be constructed out of $+1$, -1, and the integers, so it would either be hard-coded (and it must work for an infinite number of values), or produce a fixed offset from x or y. The latter case is analogous to case (1a) above, except that the branches evaluate here to a fixed offset of x or y; the same reasoning applies. In case (ii) there cannot only be $+1$ (or -1) gates for the reason outlined above. Also, there can only be a constant number of $+1$ (or -1) in a row before some $\supset_\iota$ is reached, whereupon we can apply the lemma again. By the same reasoning we must at some point encounter case (1a) of the proof.

(c) One evaluates to $pif_\iota\ b\ x\ y$ and the other to x or y. We apply the same reasoning as in case (1b) to the last gate in the branch evaluating to $pif_\iota\ b\ x\ y$, eventually reaching case (1a).

2. B is constant. That means that either N_1 or N_2 must compute $pif_\iota\ b\ x\ y$. Again we have a $\supset_\iota$ gate which does no work. Without loss of generality, assume B is tt, so N_1 always gets chosen. We apply the same reasoning as in case (1b) to the last gate in N_1 eventually reducing the problem to case (1a).

So our circuit cannot be filled with gates which "do no work." At some point there must be a $\supset_\iota$ which essentially attempts to choose between x and y. But that is impossible by Lemma 4.3.2. Therefore, our pif_ι program cannot have even this more general form. □

Now we are ready to prove the main result of this section.

Proposition 4.3.4 *PCF + por cannot implement pif_ι with a constant-depth circuit.*

Proof: Assume there exists a constant-depth circuit computing pif_ι. There are two possibilities:

1. Static circuit. The result has type integer. Therefore, there are two cases for the last gate in the circuit:

   (a) $\supset_\iota$. By Lemma 4.3.3 this is not possible.

   (b) $+1$ or -1. The circuit cannot be constructed entirely out of $+1$, -1, integers, x, y, because the result would be either hard-coded (and it must work for an infinite number of values), or a fixed offset of x or y. Also, since the circuit is static, there can only be a constant number of $+1$ or -1 in a row before reaching an occurrence of $\supset_\iota$. Then we have essentially the same situation as in case (1a) (modulo some fixed offset, as in the proof of Lemma 4.3.3), and by the same argument the circuit cannot implement pif_ι.

2. Dynamic circuit. We want to show that the circuit cannot be constant-depth. Assume, for a contradiction, that there is a fixed maximum constant depth beyond which the recursion does not get unwound, regardless of the inputs b, x, y. Then there are only finitely many constant-depth circuits which could be the result of the unwinding. But there are infinitely many possible inputs. Therefore, at least one of these circuits must work for infinitely many inputs. Apply the same reasoning on that circuit as in case (1) of this proof. We can assume there is no other recursion, otherwise continue the argument on the innermost recursion, which must exist because of the constant-depth assumption. Therefore, there is no fixed maximum depth for unwinding the recursion computing pif_ι.

In conclusion, it is impossible to write a constant-depth program using *por* to compute pif_ι, therefore *por* and pif_ι are not intensionally equivalent. □

4.3.2 Query versus pif_ι

In order to compare query to the other constructs, we need to make finer-grained distinctions than those in the previous section, and consequently, circuit semantics is no longer useful in the form we have presented it. The problem is due to the fact that we have a mixture of sequential and parallel constructs in the language and circuit semantics is an inherently parallel semantics. For a non-parallel evaluation strategy this implies that the running time of a program does not correspond closely to its circuit depth. To make the comparisons in this section we need to extend PCF's operational semantics with a notion of running time. We omit the details, but note that the results of this section only apply to evaluation strategies that are not parallel on the sequential constructs of PCF. We return to this point in the last chapter of the thesis.

To see that query is more powerful consider the implementation of an n-ary function, such as n-ary addition. Assume the existence of an addition operation $(+)$, so we can write sequential addition without having to use recursion: $add_2 \equiv \lambda xy.\ x + y$. We can implement binary addition with query as follows:

$$\begin{aligned}\text{padd} \equiv \lambda x_1 x_2.\ &\text{query } (x_1, x_2) \text{ is} \\ &\qquad (v_1, v_2) \Rightarrow v_1 + v_2\end{aligned}$$

Note that the addition of v_1 and v_2 is performed sequentially (this $+$ is sequential, not bitwise-parallel). This is not essential. What is important is that separate processes are started to evaluate the inputs. Thus, we can implement n-ary addition in depth *log* n by constructing a tree of binary additions.

So the question we are concerned with is whether pif_ι can also implement n-ary addition efficiently. The answer is no. The problem is that even though pif_ι can start parallel subcomputations to evaluate two integers, it must return one of them. There is no way to *combine* the results of the subcomputations. Only a limited amount of communication exists between the subcomputations: a check for equality of their results.

Proposition 4.3.5 *PCF + pif_ι cannot implement n-ary addition in depth log n.*

Proof: We identify a property that holds for our query program, *padd*, and show that it does not hold for programs of PCF + pif_ι. In *padd* the inputs are evaluated in parallel and the result is their sum. In PCF + pif_ι, the only parallel primitive is pif_ι so the inputs x and y must go through some pif_ι if they are to be evaluated in parallel. Suppose x goes through pif_ι after passing through some constant-depth circuit computing F and similarly for y and a function G. Then the output of the pif_ι is either $F(x)$ or $G(y)$. If either $F(x) = x + y$ or $G(y) = x + y$, then the addition was performed sequentially before the pif_ι. If the output of pif_ι goes into some constant-depth H such that $H(F(x)) = x + y$ or $H(G(y)) = x + y$ then the addition was also performed sequentially, this time after the pif_ι. So it is not possible to compute $x + y$ using pif_ι in such a way that x and y are evaluated in parallel. Therefore, a PCF + pif_ι program for n-ary addition must be of depth at least n. $\square$

As a corollary of the previous two propositions, we have the following:

Proposition 4.3.6 *PCF + por cannot implement n-ary addition in depth log n.*

In light of these results, we have the emergence of a picture of different levels of intensional expressiveness for deterministic parallel constructs: At the lowest level we have *por* and pif_o, which seem to be able to speed up only n-ary boolean functions. At the next level we have pif_ι, which can be used to speed up some integer functions. Finally, at the top level we have query, which can be used to speed up n-ary addition.

4.4 Comparing deterministic and nondeterministic query

A natural question to ask, after the results of the previous section, is whether relaxing the determinism constraint on the deterministic query gives us an increase in intensional expressiveness. This turns out to be a difficult question. In order to answer it we are forced to make some concessions: First, we consider a recursion-free version of PCF, since the recursive part does not mesh well with our interpretation for nondeterministic query. This is not that important, however, since we shall be making a connection with boolean circuits, which are not recursive. Second, we make a hardware assumption, in order to render the result of a subset of nondeterministic queries deterministic. This, of course, implies that we are not really comparing a deterministic and a nondeterministic construct, but rather two different machine models. We also need to go back and revisit the circuit semantics in order to obtain a precise correspondence between the dimensions of a circuit and the running time and work required to execute a program. We begin by introducing the modified language and nondeterministic query.

4.4.1 Recursion-free PCF

The big departure from the previous description of PCF is the lack of recursion. However, since we still need to talk about undefined (or "missing") inputs, we introduce "undefined" constants Ω^σ, one for each type σ. We also expand somewhat the set of arithmetic constants, but this is not essential. The new set of constants we will consider is:

$tt, ff : o$	$+, - : \iota \to \iota \to \iota$	
$n : \iota$	$\supset_\sigma : o \to \sigma \to \sigma \to \sigma$	$(\sigma \in \{o, \iota\})$
$=, <, >, \leq, \geq : \iota \to \iota \to o$	$\Omega^\sigma : \sigma$	(undefined elements)

4.4.2 Nondeterministic query

We drop the consistency requirement on the various outputs of a query. As an example of the programs we can write now, here is one that turns out to be important in what follows:

$$\begin{aligned} not_\perp \equiv \lambda x.\text{query } (x) \text{ is} \\ tt &\Rightarrow ff \\ ff &\Rightarrow ff \\ \_ &\Rightarrow tt \end{aligned}$$

The semantics we presented earlier for deterministic query (Figure 4.2) also makes sense for nondeterministic query, but now permits the *amb* operator to be presented with distinct defined inputs. Under this interpretation, however, programs no longer compute functions, but relations. Given the fact that our definition of intensional expressiveness requires the computation of functions, we would have to restrict the indeterminacy to the inside of a program. The following proposition shows that we cannot do that in any useful way.

Proposition 4.4.1 *Under the semantics of Figure 4.2, nondeterministic query is not intensionally more expressive than deterministic query.*

Proof: Let P be a nondeterministic program which computes the function f, and suppose it uses $amb(x, y)$, with $x \neq y$. Since P must compute a deterministic answer (by the definition of intensional expressiveness), the $amb(x, y)$ must be "determinized" somehow. Essentially, the only way that could be achieved is to throw it out: either not use it, or use it in one branch of a conditional that always chooses the other branch, or use it as the argument to a function whose result is independent of its input. But then we can certainly write a deterministic version of P that computes f with the same efficiency. $\square$

$$\mathcal{D}_{pat}[\![x^\sigma \text{ is } \__{ij}]\!]\rho = \begin{cases} (\text{not}\,(\text{poll } x^\sigma), \perp), & \text{if exhausted}(x^\sigma, i, j, p_1, \ldots, p_k, \rho) \\ (tt, \perp), & \text{otherwise} \end{cases}$$

$$\text{exhausted}(x^o, i, j, p_1, \ldots, p_k, \rho) = \begin{cases} tt, & \text{if } \begin{array}{l} ((\exists l, m.\ \mathcal{D}[\![e_{lj}]\!]\rho = tt \wedge \mathcal{D}[\![e_{mj}]\!]\rho = f\!f) \vee \\ (\exists l.\ \mathcal{D}[\![e_{lj}]\!]\rho = y^o)) \wedge (p_1\backslash j \Uparrow \cdots \Uparrow p_k\backslash j) \end{array} \\ f\!f, & \text{otherwise} \end{cases}$$

$$\text{exhausted}(x^\iota, i, j, p_1, \ldots, p_k, \rho) = \begin{cases} tt, & \text{if } (\exists l.\ \mathcal{D}[\![e_{lj}]\!]\rho = y^\iota) \wedge (p_1\backslash j \Uparrow \cdots \Uparrow p_k\backslash j) \\ f\!f, & \text{otherwise} \end{cases}$$

Figure 4.6: Semantics for nondeterministic query

We still want to restrict ourselves to programs that compute functions, even though they may use nondeterministic query, but we need a different interpretation for the meaning of nondeterministic query. One possibility is shown in Figure 4.6. The idea is to allow the "don't care" symbol to represent $\perp$ in certain circumstances, *e.g.,* when the corresponding pattern position has been exhaustively checked for all other alternatives, and the remainder of the pattern is consistent. The semantics is the same as in Figure 4.2 except for the "don't care" symbol. Let us say that such a symbol is found at location j in the i^{th} pattern (written $\__{ij}$). If the other patterns exhaustively check the input at location j and are otherwise consistent ($p\backslash j$ refers to the pattern p without location j), then the meaning of matching x^σ to "_" is a *poll* of the input. *Poll* is a nondeterministic construct [68] which checks whether an input is available:

$$poll\ \perp = f\!f, \qquad poll\ x = tt, \text{ if } x \neq \perp.$$

The p_1 through p_k are the k patterns from the general form of a query (*cf.* Figure 4.1(b)). The notation e_{lj} refers to the element at location j in the l^{th} pattern. The function *exhausted* has two cases: if the input is of type boolean, then it looks for both a tt and $f\!f$ elements in the corresponding position in the other patterns. If the input is an integer, it looks for a variable in the corresponding position (since once cannot exhaust all other integers by enumeration in the other patterns). In both the boolean and the integer case, the remainder of the pattern is checked for consistency.

Example 4.4.2 *We present examples of interpreting "don't care" as $\perp$ and others where we do not. In the following:*

$$\begin{array}{lll} tt & (tt, 1) & (1, f\!f) \\ f\!f & (f\!f, 1) & (x, f\!f) \\ \_ & (\_, 1) & (\_, f\!f), \end{array}$$

the "don't care" is interpreted as $\perp$. However, in the following two examples it is not:

$$\begin{array}{ll} (tt, 1) & (1, f\!f) \\ (f\!f, 0) & (2, f\!f) \\ (\_, 1) & (\_, f\!f). \end{array}$$

Now we can identify a subset of the nondeterministic queries which return deterministic results, assuming we can detect undefined inputs. When we have a "$\__{ij}$" interpreted as $\perp$ we can take its meaning to be $\mathcal{D}[\![\_{ij}]\!]\rho = \star$, where we have added the element $\star$ to the flat domains D_{bool}, D_{int}, with $\bot \sqsubseteq \star$. Then under the above definition of consistency we obtain the desired queries.

An example of a query which returns deterministic answers, assuming we can detect undefined inputs, is the $not_\bot$ program presented earlier. Because $\mathcal{D}[\![\_]\!]\rho = \star$, the three patterns are not consistent, and so it is fine for the result of the last pattern to be different. If we extend the $not_\bot$ query with another line, such as $\_ \Rightarrow \mathit{ff}$, then the query will no longer return deterministic answers, since both "_" symbols will be interpreted as $\bot$.

We call the extension of PCF with deterministic query DPCF, and the extension with non-deterministic query NPCF. It is quite obvious that NPCF is extensionally more expressive than DPCF, but we are interested in the following question: *Is NPCF intensionally more expressive than DPCF?* Since NPCF is an extension of DPCF it can certainly compute as efficiently as DPCF. However, to show that NPCF is more expressive, we must exhibit a function expressible in both DPCF and NPCF, and prove that DPCF cannot compute it as efficiently. This question turns out to be analogous to a problem in computational complexity theory, that of comparing monotone and De Morgan boolean circuits. Before showing why, we return to our circuit semantics.

4.5 Circuit semantics revisited

As we have seen, the basic idea of circuit semantics is very simple and has much in common with dataflow networks: view each construct of PCF as a gate, and view a computation as data flowing through the gate. The whole program becomes a circuit. Earlier we considered the circuits as being executed bottom-up. This reflected our intuition about recursion being unwound on demand. But now we wish to view them as being executed top-down, given our upcoming comparison with boolean circuits. We shall also make very precise the relationship between the dimensions of a circuit and various parallel evaluation orders for PCF, and introduce new gates to model queries.

4.5.1 Circuits for PCF

Figure 4.7(a)(b) shows the circuits representing constants and variables. The truth values and the integers are represented by nodes with no inputs. Nodes may have several outputs (fan-out is unlimited). Each of the functional constants is a node with the required number of inputs. A variable is represented by a wire, or for higher-order functions, a placeholder circuit labelled with the variable name. Figure 4.7(c) shows an input that is ignored; we need such a convention to write functions like the K combinator. The representation of conditionals in Figure 4.7(a) shows one of the essential differences with dataflow networks, which use switch and merge nodes to avoid evaluating more than one branch of the conditional during *data-driven* (top-down) execution. For *demand-driven* (bottom-up) execution of the network the difference is irrelevant. Figure 4.7(d)(e) shows the representation of lambda abstraction and application. This points out the other major difference with dataflow networks: there are no application nodes. Figure 4.8 shows the circuits representing three simple PCF terms. Note that our representation builds in sharing of arguments of ground type (but functional arguments are not shared).

Since it is rather difficult to reason about circuit dimensions in pictorial form, we define a model for the dimensions of a circuit. Figure 4.9 is an extended semantics for PCF, returning step-counting versions of a term, *S-Terms* (*cf. t-programs* in [77]), computing its depth and size. Evaluating the step-counting depth and size translations of a program gives us its depth and size. We assume renaming of bound variables, so that all identifiers are unique. The η environment in the size translation ensures sharing of arguments of ground type. Whenever a variable x of ground type

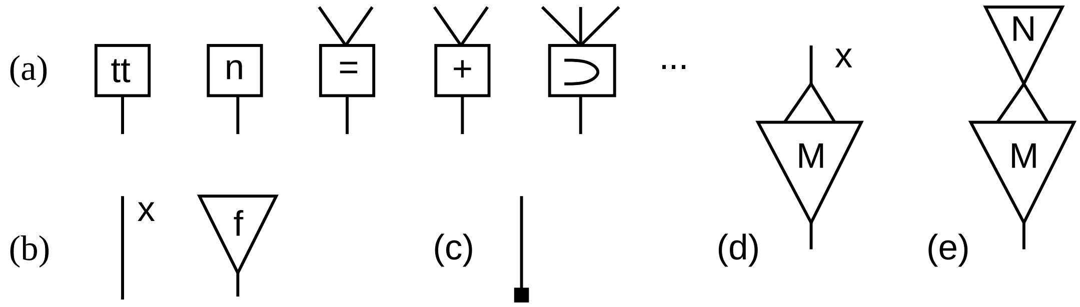

Figure 4.7: Circuits for (a) constants, (b) variables, (c) ignored inputs, (d) abstraction, (e) application, (f) query

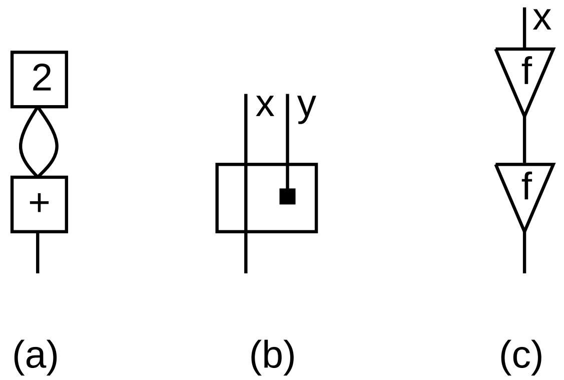

Figure 4.8: Examples: (a) $(\lambda x.\ x + x)2$, (b) $\lambda xy.\ x$, (c) $\lambda fx.\ f(fx)$

is encountered, the environment η is checked. If the variable does not occur in the environment, *i.e.*, $eta(x) = \perp$, then η is extended with the binding $x \mapsto 1$. On subsequent encounters of the same variable, its size will not be counted. Recall that we assume renaming of bound variables, in order to avoid any clashes in η. π_1 is the first projection. The syntax of S-Terms uses Λ rather than λ merely to emphasize the distinction between the programming language and the meta-language.

Example 4.5.1 *Consider the program $(\lambda x.\ x + x)2$, whose circuit semantics is depicted in Figure 4.8(a). The meaning of the program under the extended semantics is:*

$$\mathcal{E}[\![(\lambda x.\ x + x)2]\!]\perp = (4, (\Lambda x.\ 1 + \max(x, x))1, (\Lambda x.\ 1 + x + 0)1).$$

Obtaining the S-term *for the depth is straightforward. We describe in more detail how the size* S-term *is calculated:*

$$\begin{aligned} &\mathrm{size}((\lambda x.\ x + x)2)\perp = (s_1, s_2, \eta),\ \textit{where} \\ &\quad (s_1, \eta') = \mathrm{size}(\lambda x.\ x + x)\perp = (\Lambda x.\ s_1', \eta'), \\ &\quad (s_2, \eta'') = \mathrm{size}(2)\eta', \end{aligned}$$

It is in the evaluation of $(s_1', \eta') = \mathrm{size}(x + x)\perp$ that we use the environment:

$$\begin{aligned} &s_1' = \mathrm{size}(x + x)\perp = (1 + s_{11} + s_{12}, \eta_{12}),\ \textit{where} \\ &\quad (s_{11}, \eta_{11}) = size(x)\perp = (x, x \mapsto 1), \\ &\quad (s_{12}, \eta_{12}) = size(x)(x \mapsto 1) = (0, x \mapsto 1). \end{aligned}$$

From the above we deduce that:

$$\begin{aligned} &size(\lambda x.\ x + x)\perp = (\Lambda x.\ 1 + x + 0, x \mapsto 1),\ \textit{and} \\ &size(2)(x \mapsto 1) = (1, x \mapsto 1). \end{aligned}$$

Evaluating the depth and size step-counting programs, we get a depth and size of 2, which match the circuit dimensions in Figure 4.8(a).

We now prove that the circuit dimensions are indeed matched by the extended semantics.

Proposition 4.5.2 *For a PCF program M, $\mathcal{E}[\![M]\!]\perp = (v, d, s)$ if and only if the circuit representing M has depth d and size s.*

Proof: By induction on the structure of M. We need an induction hypothesis that works at higher types (*cf.* [70] for a similar example). We define predicates Match^{σ} by induction on types:

1. If M^{σ} is a program, then M^{σ} has property Match^{σ} if $\mathcal{E}[\![M^{\sigma}]\!]\perp = (v, d, s)$ if and only if the circuit representing M has depth d and size s.

2. If $M^{\sigma \to \tau}$ is a closed term, then it has property $\mathrm{Match}^{\sigma \to \tau}$ if whenever N^{σ} is a closed term with property Match^{σ}, $M^{\sigma \to \tau}\ N^{\sigma}$ has property Match^{τ}.

3. If M^{σ} is an open term with free variables $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$, then it has property Match^{σ} if the term $[N_1/x_1]\cdots[N_n/x_n]M^{\sigma}$ has property Match^{σ} whenever $N_1, \ldots, N_n$ are closed terms having properties $\mathrm{Match}^{\sigma_1}, \ldots, \mathrm{Match}^{\sigma_n}$, respectively.

$\mathcal{E}\colon \text{Terms} \rightarrow \text{Environments} \rightarrow (\bigcup D_\sigma \times \text{S-Terms} \times \text{S-Terms})$

$\mathcal{E}[\![M]\!]\rho = (\mathcal{D}[\![M]\!]\rho, \mathit{depth}(M), \pi_1(\mathit{size}(M)\bot))$, where

$$
\begin{aligned}
&\mathit{depth}\colon \text{Terms} \rightarrow \text{S-Terms}\\
&\mathit{depth}(tt) = 1\\
&\mathit{depth}(n) = 1\\
&\mathit{depth}(M_1 = M_2) = 1 + \max(\mathit{depth}(M_1), \mathit{depth}(M_2))\\
&\mathit{depth}(M_1 + M_2) = 1 + \max(\mathit{depth}(M_1), \mathit{depth}(M_2))\\
&\mathit{depth}(\supset_\iota\ M_1\ M_2\ M_3) = 1 + \max(\mathit{depth}(M_1), \mathit{depth}(M_2), \mathit{depth}(M_3))\\
&\mathit{depth}(x^\sigma) = x^\sigma\\
&\mathit{depth}(M_1\ M_2) = \mathit{depth}(M_1)\ \mathit{depth}(M_2)\\
&\mathit{depth}(\lambda x^\sigma.\ M) = \Lambda x^\sigma.\ \mathit{depth}(M)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{size}\colon \text{Terms} \rightarrow \text{Environments} \rightarrow (\text{S-Terms} \times \text{Environments})\\
&\mathit{size}(tt)\eta = (1, \eta)\\
&\mathit{size}(n)\eta = (1, \eta)\\
&\mathit{size}(M_1 = M_2)\eta = (1 + s_1 + s_2, \eta''),\ \text{where } (s_1, \eta') = \mathit{size}(M_1)\eta,\ (s_2, \eta'') = \mathit{size}(M_2)\eta'\\
&\mathit{size}(M_1 + M_2)\eta = (1 + s_1 + s_2, \eta''),\ \text{where } (s_1, \eta') = \mathit{size}(M_1)\eta,\ (s_2, \eta'') = \mathit{size}(M_2)\eta'\\
&\mathit{size}(\supset_\iota\ M_1\ M_2\ M_3)\eta = (1 + s_1 + s_2 + s_3, \eta'''),\ \text{where}\\
&\qquad\qquad (s_1, \eta') = \mathit{size}(M_1)\eta,\ (s_2, \eta'') = \mathit{size}(M_2)\eta',\ (s_3, \eta''') = \mathit{size}(M_2)\eta''\\
&\mathit{size}(x^\sigma)\eta = \begin{cases} (x^\sigma, \eta[x^\sigma \mapsto 1]), & \text{if } \sigma \in \{o, \iota\} \text{ and } \eta(x^\sigma) = \bot\\ (0, \eta), & \text{if } \sigma \in \{o, \iota\} \text{ and } \eta(x^\sigma) = 1\\ (x^\sigma, \eta), & \text{otherwise}\end{cases}\\
&\mathit{size}(M_1\ M_2)\eta = (s_1 s_2, \eta''), \text{where } (s_1, \eta') = \mathit{size}(M_1)\eta,\ (s_2, \eta'') = \mathit{size}(M_2)\eta'\\
&\mathit{size}(\lambda x^\sigma.\ M)\eta = (\Lambda x^\sigma.\ s_1, \eta'),\ \text{where } (s_1, \eta') = \mathit{size}(M)\eta
\end{aligned}
$$

Figure 4.9: Extended semantics for PCF

We call a term M^σ *matching* if it has property Match$^\sigma$. We need to prove that all terms are matching. This is obvious for boolean and integer constants. We only consider one functional constant, as the proof is similar for the others.

1. $M \equiv M_1 + M_2$. By induction hypothesis, M_1, M_2 have property Match$^\iota$. We have several cases depending on whether the terms M_1, M_2 are closed or open:

   (a) M_1, M_2 are both closed. Then by case (1) of the definition of Match$^\iota$, $\mathcal{E}[\![M_1]\!]\bot = (v_1, d_1, s_1)$ and $\mathcal{E}[\![M_2]\!]\bot = (v_2, d_2, s_2)$ iff the circuits representing M_1, M_2 have dimensions d_1, s_1 and d_2, s_2, respectively. Since both terms are closed there is no possible sharing, so by the definition of $\mathcal{E}$ and of the circuits, the depth and size of M_1+M_2 are $1+\max(d_1, d_2)$, $1 + s_1 + s_2$, respectively.

   (b) M_1 is open, M_2 is closed. By case (3) of the definition of Match$^\iota$, any closed instantiation of M_1 with matching terms will have property Match$^\iota$. Since M_2 is closed, there is no sharing between M_1 and M_2, so, as before, the dimensions match.

   (c) M_1, M_2 are both open. Consider the set $S = FV(M_1) \cap FV(M_2)$, restricted to variables of ground type. By case (3) of the induction hypothesis, any closed instantiation of M_1, M_2 with matching terms will have property Match$^\iota$. Then the depth of $M_1 + M_2$ will be the same in the two models, since sharing is not relevant. The size of $M_1 + M_2$ in the circuit model will only count each instantiation of a shared variable $x^\sigma \in S$ once. But that is exactly what the η environment is for in the $\mathcal{E}$ model. Therefore, the size will also match.

2. $M \equiv x^\sigma$. Any closed instantiation of x^σ by a term satisfying Match$^\sigma$ will have the same property.

3. $M \equiv M_1^{\sigma\to\tau} M_2^\sigma$. By the induction hypothesis, $M_1^{\sigma\to\tau}$ satisfies property Match$^{\sigma\to\tau}$ and M_2^σ has property Match$^\sigma$. If both M_1, M_2 are closed, then by case (2) of the induction hypothesis, $M_1^{\sigma\to\tau} M_2^\sigma$ has property Match$^\tau$. If M_1, M_2 are open, then construct the set S as above. Any closed instantiation of M_1, M_2 with matching terms will result in the sharing of $x \in S$, both in circuits and in the $\mathcal{E}$ model. Therefore $M_1^{\sigma\to\tau} M_2^\sigma$ will have property Match$^\tau$.

4. $M \equiv \lambda x^\sigma.\ M^\tau$. By the induction hypothesis, M^τ has property Match$^\tau$. We need to show that $\lambda x^\sigma.\ M^\tau$ has property Match$^{\sigma\to\tau}$. Suppose that $\lambda x^\sigma.\ M^\tau$ is closed. If $\sigma \in \{o, \iota\}$, then for any input N^σ both the circuits and the $\mathcal{E}$ model will only include one copy of N^σ. The depth function from the $\mathcal{E}$ model actually does no sharing, but this is irrelevant, as the depth is unaffected. Then $(\lambda x^\sigma.\ M^\tau)N^\sigma$ will have property Match$^\tau$. For general σ, there are no sharing issues, so the result again has property Match$^\tau$. Now suppose $\lambda x^\sigma.\ M^\tau$ is open. Any closed instantiation of $\lambda x^\sigma.\ M^\tau$ with matching terms will result in the same sharing of common variables in both circuits and the $\mathcal{E}$ model. Therefore, $\lambda x^\sigma.\ M^\tau$ has property Match$^{\sigma\to\tau}$.

Since all M^σ have property Match$^\sigma$, certainly the programs enjoy this property as well, which by case (1) of the definition of Match$^\sigma$ establishes our proposition. □

Before undertaking a comparison of our circuit model with various evaluation strategies, we discuss the issues involved. Consider the following PCF program: $P \equiv \supset_\iota tt\ 2\ M$, where M is an expression whose evaluation takes many steps before returning an integer. The circuit representation of P will include a piece corresponding to M. But in call-by-name PCF (or call-by-need, a particular implementation of call-by-name), M will not be evaluated, so the size of the circuit representing P will be a wild overestimate. Of course, we could evaluate the circuit bottom-up, thus avoiding evaluation of M, but this would preclude any meaningful discussion of circuit size or depth. The problem is that circuits are most closely related to dataflow networks, which, in turn, are most naturally implemented in a data-driven fashion, an embodiment of call-by-value. Even though call-by-name (call-by-need) can also be implemented in parallel using graph reduction, that model is basically the equivalent of upside-down demand-driven evaluation of a dataflow network. Thus it seems reasonable to confine ourselves to comparisons with evaluation strategies which are natural for dataflow networks.

We compare our circuit model to two different evaluation strategies: parallel call-by-value (c-b-v) [49, 9] and call-by-speculation (c-b-s) [49, 75, 41]. Given an application $f\ x$, in parallel c-b-v the function f and the argument x are evaluated in parallel and after *both* evaluations are completed, then the body of the function f is evaluated. Therefore, parallel c-b-v takes advantage of *horizontal parallelism* (evaluating two tasks simultaneously), and also to a lesser extent of *vertical parallelism* (pipelining).

In c-b-s, we also evaluate f and x in parallel, but we do not require that the evaluation of x be complete before we evaluate the body of f. Thus c-b-s allows fully pipelined parallelism. If x gives rise to a large computation that is not used by f we will get a result much faster, but the computation will still continue for some time. C-b-s thus introduces a distinction between *minimum* and *maximum* time to evaluate an expression.

Figure 4.10 shows a profiling semantics for parallel c-b-v in the style of [9]. Judgments have the form $\rho \vdash M \longrightarrow_{\mathrm{cbv}} v; d, w$, meaning that in environment ρ, M evaluates to v in d steps (depth) and with w work. The possible results of an evaluation are values, ranged over by v, and they are either constants or function closures:

$$v ::= c \mid cl(\rho, x, M).$$

The rule for addition is typical of the treatment of most constructs, in that the depth of the computation is the *maximum* of the depths of the subcomputations, with the addition of a constant for the evaluation of the construct itself. The size of of the computation is the *sum* of the subcomputation plus a constant. In the case of conditionals, the condition is evaluated first, and then the appropriate branch is chosen. Finally, the rule for application shows that evaluation of the function body waits for evaluation of the argument to complete.

The constants chosen for the depth and work in several of the rules are different from [9], as we want to achieve an exact match with our circuit semantics. The differences are not significant.

The correspondence between our circuit semantics and parallel c-b-v is fairly simple: as long as we have conditionals in a program, the models are incomparable, for the reasons mentioned earlier. However, without conditionals, circuits will take less time and do less work than predicted by the operational semantics, in the following sense:

Proposition 4.5.3 *Given a PCF program M with $\mathcal{E}[\![M]\!]\bot = (v, d, s)$, and $\bot \vdash M \longrightarrow_{\mathrm{cbv}} v; t, w$, the following hold:*

1. *If M is conditional-free, then $s \leq w$;*

2. *If M is conditional-free, then $d \leq t$.*

Proof: By induction on M as in the proof of Proposition 4.5.2. We define predicates Val^{σ} by induction on types:

$$\rho \vdash tt \longrightarrow_{\mathrm{cbv}} tt; 1, 1$$

$$\frac{\rho \vdash M_1 \longrightarrow_{\mathrm{cbv}} v_1; d_1, w_1 \qquad \rho \vdash M_2 \longrightarrow_{\mathrm{cbv}} v_2; d_2, w_2}{\rho \vdash M_1 + M_2 \longrightarrow_{\mathrm{cbv}} v_1 + v_2; 1 + \max(d_1, d_2), 1 + w_1 + w_2}$$

$$\frac{\rho \vdash M_1 \longrightarrow_{\mathrm{cbv}} tt; d_1, w_1 \qquad \rho \vdash M_2 \longrightarrow_{\mathrm{cbv}} v_2; d_2, w_2}{\rho \vdash \supset_\iota M_1 M_2 M_3 \longrightarrow_{\mathrm{cbv}} v_2; 1 + d_1 + d_2, 1 + w_1 + w_2}$$

$$\frac{\rho \vdash M_1 \longrightarrow_{\mathrm{cbv}} f\!f; d_1, w_1 \qquad \rho \vdash M_3 \longrightarrow_{\mathrm{cbv}} v_3; d_3, w_3}{\rho \vdash \supset_\iota M_1 M_2 M_3 \longrightarrow_{\mathrm{cbv}} v_3; 1 + d_1 + d_3, 1 + w_1 + w_3}$$

$$\frac{\rho(x) = v}{\rho \vdash x \longrightarrow_{\mathrm{cbv}} v; 0, 0}$$

$$\rho \vdash \lambda x.\, M \longrightarrow_{\mathrm{cbv}} cl(\rho, x, M); 0, 0$$

$$\frac{\begin{array}{l}\rho \vdash M_1 \longrightarrow_{\mathrm{cbv}} cl(\rho', x, M_1'); d_1, w_1 \\ \rho \vdash M_2 \longrightarrow_{\mathrm{cbv}} v_2; d_2, w_2\end{array} \qquad \rho'[x/v_2] \vdash M_1' \longrightarrow_{\mathrm{cbv}} v; d_3, w_3}{\rho \vdash M_1\ M_2 \longrightarrow_{\mathrm{cbv}} v; \max(d_1, d_2) + d_3, w_1 + w_2 + w_3}$$

Figure 4.10: Profiling semantics for parallel call-by-value

1. If M^σ is a program, then M^σ has property Val^σ if $\mathcal{E}[\![M^\sigma]\!]\bot = (v, d, s)$ if and only if $\bot \vdash M \longrightarrow_{\text{cbv}} v; t, w$, with $s \leq w, d \leq t$.

2. If $M^{\sigma \to \tau}$ is a closed term, then it has property $\text{Val}^{\sigma\to\tau}$ if whenever N^σ is a closed term with property Val^σ, $M^{\sigma\to\tau}\ N^\sigma$ has property Val^τ.

3. If M^σ is an open term with free variables $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$, then it has property Val^σ if the instantiation $[N_1/x_1]\cdots[N_n/x_n]M^\sigma$ has property Val^σ whenever $N_1, \ldots, N_n$ are closed terms having properties $\text{Val}^{\sigma_1}, \ldots, \text{Val}^{\sigma_n}$, respectively.

The result is easily verifiable for constants (except, of course, for conditionals), so we only consider the induction step:

1. $M \equiv x^\sigma$. Any closed instantiation of x^σ by a term satisfying Val^σ will have the same property.

2. $M \equiv M_1^{\sigma\to\tau} M_2^\sigma$. By the induction hypothesis, $M_1^{\sigma\to\tau}$ has property $\text{Val}^{\sigma\to\tau}$ and M_2^σ has property Val^σ. If both M_1, M_2 are closed, then by case (2) of the induction hypothesis, $M_1^{\sigma\to\tau} M_2^\sigma$ has property Val^τ. If M_1, M_2 are open, then construct the set $S = FV(M_1) \cap FV(M_2)$, restricted to variables of ground type. Any closed instantiation of M_1, M_2 with closed terms satisfying Val will result in the sharing of $x^{\in}S$, in both circuits and parallel c-b-v. Therefore $M_1^{\sigma\to\tau} M_2^\sigma$ will have property Val^τ.

3. $M \equiv \lambda x^\sigma.\ M^\tau$. We need to show that $\lambda x^\sigma.\ M^\tau$ has property $\text{Val}^{\sigma\to\tau}$. By the induction hypothesis, M^τ has property Val^τ. Suppose that $\lambda x^\sigma.\ M^\tau$ is closed. Let N^σ be a closed term satisfying Val^σ. If M^τ is closed (*i.e.*, x^σ is not used in M^τ), then $\mathit{depth}((\lambda x^\sigma.\ M^\tau)N^\sigma) = \mathit{depth}(M^\tau)$ and similarly for size. Since M^τ has property Val^τ, so must $(\lambda x^\sigma.\ M^\tau)N^\sigma$ (parallel c-b-v requires more time and work to evaluate $(\lambda x^\sigma.\ M^\tau)N^\sigma$ than M^τ). This is the origin of the inequalities in the proposition: parallel c-b-v will evaluate unused arguments.

   If M^τ is open, then if $\sigma \in \{o, \iota\}$, N^σ will be shared in both circuits and parallel c-b-v, so the depth and work will be the same. For general σ, by the third part of the induction hypothesis, $[N^\sigma/x^\sigma]M^\tau$ will have property Val^τ.

   Now suppose $\lambda x^\sigma.\ M^\tau$ is open. Any closed instantiation of $\lambda x^\sigma.\ M^\tau$ with closed terms satisfying Val will result in the same sharing of common variables of ground type in both circuits and parallel c-b-v. Therefore, $\lambda x^\sigma.\ M^\tau$ has property $\text{Val}^{\sigma\to\tau}$.

Since all terms M^σ have property Val^σ, so do the programs, which establishes our result. □

Figure 4.11 shows a profiling semantics for c-b-s, in the style of [41]. Because of the distinction between minimum and maximum depth, the judgments are different from the parallel c-b-v ones. They are of the form

$$\rho, d \vdash M \longrightarrow_{\text{cbs}} v; d', \hat{d}', w,$$

meaning that in environment ρ, evaluating M at depth d leads to result v which will be available at depth d', while the whole computation will be done at depth $\hat{d}'$ and use w work. In addition, the environment ρ has to keep track at which depth a value becomes available. This is equivalent to the effect achieved in Roe's semantics [75] with time stamps.

The rules for constant, addition, and abstraction are not significantly different from the parallel c-b-v case, because minimum and maximum depth are the same. The rule for variables takes into account the depth at which a value becomes available. The rules for conditionals, show that minimum depth is the same as before, but maximum depth incorporates the evaluation of all

$$\rho, d \vdash tt \longrightarrow_{\text{cbs}} tt; d+1, d+1, 1$$

$$\frac{\rho, d \vdash M_1 \longrightarrow_{\text{cbs}} v_1; d_1, \hat{d}_1, w_1 \qquad \rho, d \vdash M_2 \longrightarrow_{\text{cbs}} v_2; d_2, \hat{d}_2, w_2}{\rho, d \vdash M_1 + M_2 \longrightarrow_{\text{cbs}} v_1 + v_2; 1 + \max(d_1, d_2), 1 + \max(\hat{d}_1, \hat{d}_2), 1 + w_1 + w_2}$$

$$\frac{\begin{array}{l}\rho, d \vdash M_1 \longrightarrow_{\text{cbs}} tt; d_1, \hat{d}_1, w_1 \\ \rho, d \vdash M_2 \longrightarrow_{\text{cbs}} v_2; d_2, \hat{d}_2, w_2\end{array} \qquad \rho, d \vdash M_3 \longrightarrow_{\text{cbs}} v_3; d_3, \hat{d}_3, w_3}{\rho, d \vdash \supset_\iota M_1 M_2 M_3 \longrightarrow_{\text{cbs}} v_2; 1 + \max(d_1, d_2), 1 + \max(\hat{d}_1, \hat{d}_2, \hat{d}_3), 1 + w_1 + w_2 + w_3}$$

$$\frac{\begin{array}{l}\rho, d \vdash M_1 \longrightarrow_{\text{cbs}} ff; d_1, \hat{d}_1, w_1 \\ \rho, d \vdash M_2 \longrightarrow_{\text{cbs}} v_2; d_2, \hat{d}_2, w_2\end{array} \qquad \rho, d \vdash M_3 \longrightarrow_{\text{cbs}} v_3; d_3, \hat{d}_3, w_3}{\rho, d \vdash \supset_\iota M_1 M_2 M_3 \longrightarrow_{\text{cbs}} v_3; 1 + \max(d_1, d_3), 1 + \max(\hat{d}_1, \hat{d}_2, \hat{d}_3), 1 + w_1 + w_2 + w_3}$$

$$\frac{\rho(x) = v; d'}{\rho, d \vdash x \longrightarrow_{\text{cbs}} v; \max(d, d'), \max(d, d'), 0}$$

$$\rho, d \vdash \lambda x.\ M \longrightarrow_{\text{cbs}} cl(\rho, x, M); d, d, 0$$

$$\frac{\begin{array}{l}\rho, d \vdash M_1 \longrightarrow_{\text{cbs}} cl(\rho', x, M_1'); d_1, \hat{d}_1, w_1 \\ \rho, d \vdash M_2 \longrightarrow_{\text{cbs}} v_2; d_2, \hat{d}_2, w_2\end{array} \qquad \rho'[x/v_2; d_2], d_1 \vdash M_1' \longrightarrow_{\text{cbs}} v; d_3, \hat{d}_3, w_3}{\rho, d \vdash M_1\ M_2 \longrightarrow_{\text{cbs}} v; d_3, \max(\hat{d}_1, \hat{d}_2, \hat{d}_3), w_1 + w_2 + w_3}$$

Figure 4.11: Profiling semantics for call-by-speculation

branches. Similarly, the maximum depth of an application includes the evaluation of the function, the argument, and of the function body, while the minimum depth will only include the evaluation of the argument if it is used.

It should be fairly clear by now that the circuit semantics is very closely related to c-b-s. In fact, it would be a slight improvement on it were it not for conditionals. A circuit will always do less work than c-b-s, but its depth might be larger than the c-b-s minimum depth. In a conditional-free program P, however, the depth of the circuit semantics of P is the same as the c-b-s minimum depth.

Proposition 4.5.4 *If $\mathcal{E}[\![M]\!]\bot = (v, d, s)$, and $\bot, 0 \vdash M \longrightarrow_{\text{cbs}} v'; t, \hat{t}, w$, then $v = v'$ and we have: (1) $s \leq w$; (2) $d \leq \hat{t}$; (3) if M is conditional-free, then $d = t$.*

Proof: Similar to the proof of Proposition 4.5.3. □

There is another order of evaluation to which we could compare our circuit model. Hudak and Anderson [49] consider parallel *eager* evaluation, a version of parallel c-b-v. Instead of requiring the argument to complete evaluating before the body of the function is evaluated, we require that it complete before the function call returns. This gives extra parallelism over parallel c-b-v, but might still evaluate some arguments unnecessarily. The correspondence between parallel eager evaluation and circuits would be the same as that between parallel c-b-v and circuits.

4.5.2 Circuits for query

The circuit semantics of query is a straightforward implementation of the semantics presented earlier. Accordingly, we need to expand our set of basic gates with those of Figure 4.7(f). An "if-then" gate is just shorthand for an if-then-else statement, with a $\bot$ "else" part. *And* is parallel-and. For space reasons *amb* is depicted at times with more than two inputs; the understanding is that a k-ary *amb* gate stands for a balanced tree of binary *amb* gates.

Figure 4.12(a) shows the circuit semantics of parallel-or. The *not* gate is shorthand for $\supset_o x\ f\!f\ tt$. Figure 4.12(b) shows the circuit semantics of $not_\bot$ from Section 5.2, which essentially performs negation with respect to $\bot$ (note how "_" is used to represent $\bot$).

4.6 On the method and the metric

This section describes our methodology for comparing DPCF and NPCF. First, we refine the notion of intensional expressiveness to take into account both measures of program complexity induced by the circuit semantics: work and time. Then we compare our circuit model with the traditional parallel computation model, the PRAM, establishing a simple connection via an analogue of Brent's theorem. Finally, we detail the hardware assumption we make that allows us to compare nondeterministic and deterministic programs.

4.6.1 Intensional expressiveness and parallel complexity

Given a program P, the circuit semantics of P provides a definition of its parallel complexity: the parallel time and the parallel work required to execute it. Accordingly, we can define two notions of intensional expressiveness.

We say that language L_1 is *intensionally more work-expressive* than L_2 ($L_1 \succeq_w L_2$), if any function computable by L_2 with a program whose size under the circuit semantics is s_2 can be

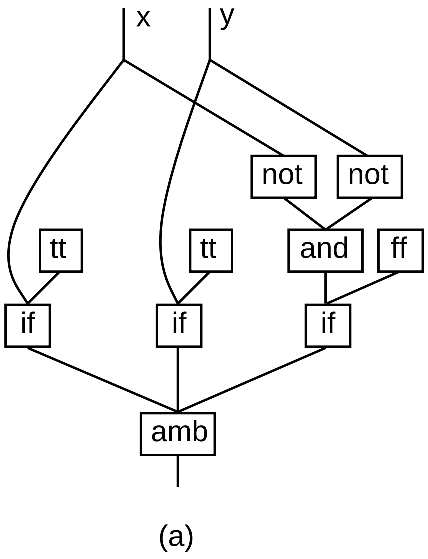

Figure 4.12: Circuit semantics for: (a) *por*, (b) $not_\perp$

computed by an L_1 program with circuit size $s_1 \leq s_2$. Similarly, L_1 is *intensionally more time-expressive* than L_2 ($L_1 \succeq_t L_2$), if any function computable by L_2 with a program whose depth under the circuit semantics is d_2 can be computed by an L_1 program with circuit depth $d_1 \leq d_2$.

As mentioned in the introduction, we are interested in asymptotic complexity: for a function f of n arguments, we want to compare the size and depth of circuits computing f as functions of n. In particular, we would like to establish separation results of the type $L_1 \succ_w L_2$ and $L_1 \succ_t L_2$.

4.6.2 Comparison with the PRAM

We compare our parallel complexity model, circuit semantics, with the standard model from the theory of parallel algorithms, the PRAM [23]. This comparison will be needed later when we establish a connection between PCF programs and boolean circuits.

Suppose we have a program M whose parallel complexity according to the circuit semantics is size s and depth d. The question we have to address is how long it takes to execute M on a p-processor PRAM.

We can imagine the process of executing M in two parts: construction and execution. First we construct the circuit semantics of M, then we execute it. We consider the execution part first, since it is very simple.

Proposition 4.6.1 *[Execution] Given the circuit semantics of program M, of size s and depth d, it can be simulated on a p-processor CREW PRAM in $O(s/p + d)$.*

Proof: This follows from Brent's theorem [23], which establishes bounds for simulating boolean circuits on the PRAM. Our circuits satisfy the necessary conditions for Brent's theorem: we have bounded fan-in, and we assume that each gate can be simulated in $O(1)$ time. $\square$

For our purposes, we do not need a Construction proposition. Rather, we want to know if there is any difference in construction time between DPCF and NPCF programs. Since the only difference is in the treatment of queries, we only consider the time needed to generate circuits for each of the two kinds of queries.

Proposition 4.6.2 *[Construction equivalence] It takes asymptotically the same time on a p-processor PRAM to construct circuits for deterministic and nondeterministic interpretations of a query.*

Proof: To implement deterministic query we need to look at every element of each pattern at least once. Then the time required to process a query on n variables and containing k patterns be must be at least $O(nk/p)$.

For the nondeterministic interpretation, we have to check if any "_" is supposed to be $\perp$. This can be achieved as follows: First check if any column in the pattern exhaustively checks for all possibilities, and identify all "_" that occur in those columns. Also, keep track if the column consists entirely of variables and "_". This can be done in $O(nk/p)$.

If all columns contain only variables and "_", then all "_" from exhaustively checked columns are $\perp$, and we are done. Otherwise, pick one column, call it i, that is exhaustively checked with more than variables and "_", and verify the rest of the pattern for consistency. This also takes $O(nk/p)$.

If consistent, then all "_" from column i are $\perp$, and we are done, since including column i in any future consistency check will fail. If not consistent, then again we are done for the same reason.

Since we only take $O(nk/p)$ to find out which "_" mean $\perp$, it takes the same time to generate circuits for both deterministic and nondeterministic query. □

4.6.3 How to compare determinism and nondeterminism

Our notion of intensional expressiveness involves comparing the complexity of computing *functions*, so we have to make sure our nondeterministic programs return deterministic values. Since a purely nondeterministic interpretation of query fails to be intensionally more expressive (Proposition 4.4.1), we chose an interpretation that allows us to get deterministic answers, assuming we can detect undefined inputs. This section discusses our assumption in more detail.

There has been a considerable amount of research in the area of algorithms for unreliable distributed systems. For instance, a whole field is devoted to consensus problems in such systems [34]. In such algorithms, a key assumption is that one can detect the failure of a process to send a message. Without this assumption, *i.e.,* in a fully asynchronous model, the consensus problem is not solvable. The physical realization of this assumption is quite reasonable, and does not even require fully synchronous hardware. Lamport [58] has shown how to detect failure to send messages through the use of timeouts. This requires a model with accurate clocks and bounds on message transit times.

We shall assume that we are able to detect undefined inputs through the use of such hardware. Since DPCF cannot take advantage of this (by the definition of deterministic query), in a sense, our question has become: Does the ability to detect undefined inputs in NPCF imply that NPCF $\succ_w$ DPCF and/or NPCF $\succ_t$ DPCF?

The difficulty in answering this question stems from the fact that we have to exhibit a function computable by *both* NPCF and DPCF, but which can be computed faster or with less work by NPCF. It is not enough to say that NPCF can compute generalized functions because they have no deterministic counterpart. On the other hand, DPCF is quite powerful. It can express parallel-or and parallel-and, so it can implement comparators for boolean values which work with undefined inputs. Using comparators we can implement asymptotically very efficient sorting networks, such as the AKS network [3], which can sort n inputs in depth $O(\lg n)$ and size $O(n \lg n)$. Since a DPCF program must output a single value we cannot just implement the sorting network, but we can use it internally, for instance, to compute the majority function, by first sorting the input and then picking the middle value [87].

Such considerations led us to examine more closely the similarity between DPCF and NPCF programs and boolean circuits. It turns out that there is a very close correspondence.

4.7 A connection with boolean circuits

This section contains a formal connection between our DPCF and NPCF languages and boolean circuits. Since there are different kinds of boolean circuits in the literature, depending on the basis, we first give a very brief overview of the circuits we are interested in.

4.7.1 Boolean circuits

A boolean circuit computes a boolean function $f : \{0,1\}^n \to \{0,1\}$. For our purposes, a boolean circuit is a directed acyclic graph whose inputs are sources of the graph, and whose nodes (gates) are selected from a set called the basis. The fan-out of the inputs and gates in the circuit is unbounded. The fan-in is 2. Normally, a circuit can have more than one output, but we are only interested in single-output circuits, since our DPCF and NPCF programs only return one output.

Gates in the basis are single output. Among the many bases studied, two are of interest to us: the *monotone* basis $\{\wedge, \vee\}$, and the De Morgan basis $\{\wedge, \vee, \neg\}$. The monotone basis is not complete, i.e. not every boolean function is computable by monotone circuits (only the monotone functions are), but it is of particular interest to researchers in complexity theory, since strong lower bounds have been obtained for monotone circuits [87]. The De Morgan basis is complete.

There are two measures for the efficiency of a circuit. The *size* or *complexity* is the number of gates in the circuit; this intuitively measures the work needed to compute the output. The *depth* is the number of gates in the circuit on the longest path from the input to any output; this measures the time required to compute the output.

There are several complexity classes defined in terms of circuits. We define those which will be used in what follows (*cf.* [10]). A *family* of circuits is a sequence $(C_1, C_2, \ldots)$, where C_n takes n input variables. A *uniform family* is one where the description of C_n can easily be computed from n. The classes NC^k (AC^k), for $k \geq 0$, consist of all functions computable by a uniform family of polynomial size, $O(\log^k n)$ depth circuits with constant (unbounded) fan-in. $\mathrm{NC} = \bigcup_{k \geq 0} \mathrm{NC}^k$, and similarly $\mathrm{AC} = \bigcup_{k \geq 0} \mathrm{AC}^k$. It is easy to show that for all $k \geq 0$, $\mathrm{AC}^k \subseteq \mathrm{NC}^{k+1} \subseteq \mathrm{AC}^{k+1}$, therefore $\mathrm{NC} = \mathrm{AC}$.

4.7.2 The connection

The main idea of the connection is very simple and can best be described by a picture. Figure 4.13 shows the domain of booleans for circuits, and the domains of booleans and natural numbers for DPCF and NPCF. Booleans are ordered differently in the circuit world than in programming language semantics. Monotone means the same thing in both places, but with respect to a different ordering. That is why negation is not monotone in the case of circuits, but it certainly is expressible in PCF, which only computes continuous (hence monotone) functions. Therefore, 0 has the same rôle in the circuit world that $\perp$ has for PCF programs.

When a monotone circuit is presented with an input, there is no way for the circuit to determine if that input is 0, since all it can do is AND and OR the inputs. This is really the same situation

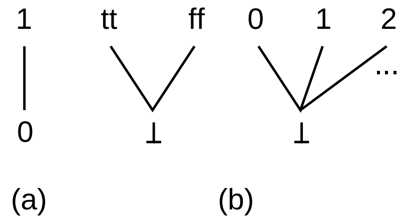

Figure 4.13: Domains for (a) boolean circuits, (b) DPCF and NPCF programs

as when a DPCF program can have undefined inputs. The DPCF constructs cannot help identify undefined inputs.

The effect of the hardware assumption we made is now clear. NPCF can detect undefined inputs, so it essentially has negation with respect to bottom (*cf.* Figure 4.12(b)). Therefore, NPCF resembles De Morgan circuits.

Since DPCF and NPCF operate on a different domain than boolean circuits, we need a way to take a function computed by a boolean circuit, say $f: \{0,1\}^n \to \{0,1\}$ and view it as a function on PCF domain elements, say $\perp, tt$ (the choice of tt rather than ff is arbitrary). Let us call the equivalent of f in the PCF world f_{pcf}. Conversely, given g: $\{\perp, tt\}^n \to \{\perp, tt\}$ in PCF, we define g_{bool} to be its counterpart in the circuit world.

DPCF and NPCF are at least as powerful as boolean circuits. Given a circuit, we can divide it into "blocks," each implementable by a DPCF (NPCF) function, and select a suitable application order that does not duplicate work, thus constructing an equivalent program. For the reasons listed above, monotone circuits are as powerful as DPCF when presented with undefined inputs.

Proposition 4.7.1 *Given a monotone (De Morgan) circuit computing f we can construct from it a DPCF (NPCF) program computing f_{pcf}, with the same dimensions in the circuit semantics.*

Proof: We only discuss the monotone case here as the other case is similar.

It is clear that the only problems we might encounter occur when the fan-out is larger than 1 for some nodes. If the fan-out is always 1, the circuit is a tree, and can be represented by a formula. If the fan-out is larger than 1 for some node, then we have to be careful we do not duplicate work in the DPCF program. Since a link in the circuit is equivalent to an application in DPCF, we want to figure out an application order such that in the circuit semantics we end up with the same structure as the original circuit.

We give an algorithm to construct the DPCF program. First, we identify all nodes with fan-out > 1 in the circuit. Then, beginning at the bottom, we identify all the parts of the circuit which are trees. That is, we go as far as possible up the circuit in all directions, until we get to the recipient of a multiple output. This determines one DPCF function. We continue in similar fashion from the origins of the multiple outputs. We have now divided the circuit into blocks, and all we need to do is figure out an application order.

We start with the last block in the circuit (the one containing the output node); call it F_0. Let N be the set of all of its immediate neighbors. F_0 will be a function of $|N|$ arguments. We apply F_0 to all elements of N from which there is no path going into another element of N. This works because there are no cycles in the circuit. We continue like this with all elements from N which were used, and which receive inputs. A minor issue is the ordering of the inputs when doing the

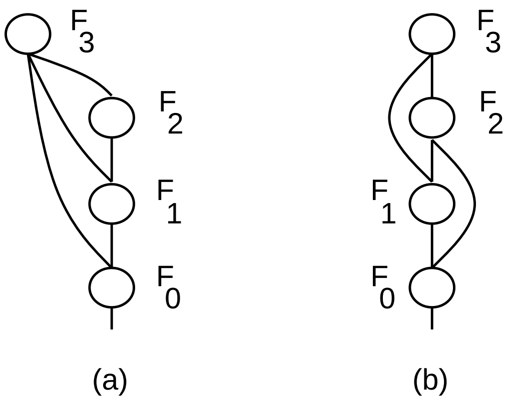

Figure 4.14: Two circuits

applications. We need to order them so that the element of N for which there exists the longest path into F_0 is the last input. In case of ties the order is not important.

We also have to make sure the variable names are the same in different blocks when they have to be bound to the same input. For instance, consider the following case: we have three nodes, F_0, F_1, and F_2, with F_1 having a link into F_0, and F_2 having links into both F_0 and F_1. Then the nodes will have the form $F_0 \equiv \lambda xy.\ f_0$, $F_1 \equiv \lambda z.\ f_1$, and the application order will be $F_0 F_1 F_2$. When we "hook up" F_0 and F_1 we have to change all the z's in F_1 to y's. Equivalently, we could have $F_1 \equiv \lambda y.\ f_1$ and use a β substitution rule that allows variable capture.

This procedure yields a DPCF program whose circuit semantics quite obviously looks the same as the original circuit, and therefore has the same size and depth. $\square$

Example 4.7.2 *We give an example to illustrate the algorithm described above. Figure 4.14 shows the block structure of two circuits. Case (a) would give rise to the following program P_a (for simplicity, we assume β substitution with variable capture, as discussed above):*

$$F_0 \equiv \lambda xy.\ f_0, \quad F_1 \equiv \lambda zx.\ f_1, \quad F_2 \equiv \lambda x.\ f_2$$
$$P_a \equiv F_0(F_1 F_2)F_3.$$

Case (b) would result in the construction of the following P_b:

$$F_0 \equiv \lambda xy.\ f_0, \quad F_1 \equiv \lambda yz.\ f_1, \quad F_2 \equiv \lambda y.\ f_2$$
$$P_b \equiv F_0 F_1 F_2 F_3.$$

Now we show that monotone circuits are as powerful as DPCF for a certain class of functions. For our applications, we do not need the equivalent statement for De Morgan circuits and NPCF programs.

Proposition 4.7.3 *If DPCF can compute $f : \{\bot, tt\}^n \to tt$, then monotone circuits can compute f_{bool} with a circuit whose dimensions are the same as the circuit semantics of the DPCF program.*

Proof: Let us assume the function f is not constant, otherwise the proof is trivial. We can divide the constructs of DPCF into sequential and parallel. There is only one parallel construct, query.

The sequential constructs, if presented with an undefined input, will all return $\perp$. Therefore, in order to obtain any result when computing f we must use query.

We can translate the *tt* in the input to anything else we would like: *ff*, any integer. We cannot translate $\perp$. The functions computable in DPCF on $\perp$ and any other non-bottom input are the same: AND, OR, constant functions. Since we must use query to make any progress, and since queries on $\perp$ and any other input are all the same, it doesn't matter what the other input is. So translating the *tt* from the input makes no difference. We would simply have to translate back at the output.

We can then eliminate the unnecessary translations and leave only the queries. The circuit semantics of the resulting program with $\perp$, *tt* inputs changed to 0,1 inputs respectively, is a valid monotone circuit computing f_{bool}. $\square$

Note that we can consider the DPCF program for f to be conditional-free, therefore, according to Propositions 4.5.3,4.5.4, the result implies that a boolean circuit can compute f_{bool} with at least the same efficiency that DPCF can compute f under either parallel c-b-v or c-b-s.

4.8 Applications

Given the correspondence we have established between DPCF, NPCF, and boolean circuits, we can use strong results from complexity theory to prove equivalent statements about our programming languages. These results are applicable for either parallel c-b-v or c-b-s (*cf.* discussion at the end of previous section), and also contain implications about evaluation on the PRAM.

Improving on an earlier superpolynomial bound by Razborov, Tardos [84] proved a very strong separation result between monotone and De Morgan circuit complexity:

Theorem 4.8.1 *[Tardos] There exists a polynomial time computable monotone function whose monotone complexity is exponential.*

The monotone function discussed is the perfect matching function [23], which takes an adjacency matrix as input and returns 1 if the graph has a perfect matching. DPCF and NPCF programs would compute the same function on $\{\perp, tt\}^n$ inputs.

Since perfect matching is in P, there exists a De Morgan circuit of polynomial size for it [10]. By Proposition 4.7.1 there is an NPCF program for it which does a polynomial amount of work. Since the monotone complexity of perfect matching is exponential, by Proposition 4.7.3 any DPCF program for it will do an exponential amount of work. Therefore, we have:

Proposition 4.8.2 *There exists a function computable by an NPCF program with polynomial work, but for which the best DPCF program does exponential work.*

By Propositions 4.6.1,4.6.2, the result can be stated in term of execution time on the PRAM: there exists a function computable by both DPCF and NPCF but which requires exponentially more time to execute on a PRAM for DPCF than NPCF.

A strong separation result between monotone and De Morgan circuits is also known for circuit depth [72].

Theorem 4.8.3 *[Raz & Wigderson] There is a monotone function in NC^1 that has no monotone NC circuits.*

The function referred to in Theorem 4.8.3 is a variant of the matching function (matching of size $n/3$, where n is the number of vertices in the graph). By a similar argument as above we can apply this result to NPCF and DPCF programs.

Proposition 4.8.4 *There exists a function computable by an NPCF program in logarithmic time, and for which the best DPCF program takes more than polylogarithmic time.*

4.9 Discussion

We have defined a new, *intensional* denotational semantics for functional languages, circuit semantics. Circuit semantics associates a gate with each basic construct of the language, and takes the meaning of a program to be a circuit. The dimensions of the circuit enable reasoning about running time and work required for execution. We have established circuit semantics as a parallel complexity model, by comparing it to the time and work required to execute programs under several parallel evaluation strategies, and in the PRAM model. We have also used circuit semantics to obtain relative intensional expressiveness results for parallel extensions of PCF.

We have shown that deterministic query is intensionally more expressive than pif_ι, which, in turn, is intensionally more expressive than *por* and pif_o. Thus, we have the beginnings of a hierarchy of intensional expressiveness for deterministic parallelism. In the process, we have exhibited languages which are extensionally but not intensionally equivalent. The constructs *por*, pif_o, and pif_ι are interdefinable in the continuous function model of PCF. However, PCF + pif_ι is intensionally more expressive than PCF + *por* (or pif_o). A natural question raised by this is whether there exists a language that is extensionally more expressive but intensionally less expressive (on the common subset of computable functions) than another language. The case of the Girard-Reynolds system F versus Gödel's system T might be an example of this, but the matter is not settled yet (*cf.* [20]).

In order to compare deterministic and nondeterministic query, we were forced to make an assumption about having the ability to detect undefined inputs. Though somewhat unaesthetic, this assumption allowed us to view the question as a similar one from complexity theory, that of comparing monotone and De Morgan boolean circuits. After establishing a connection between the dimensions of a program under the circuit semantics and the complexity and depth of a corresponding boolean function, we were able to show that nondeterministic query is intensionally more expressive than its deterministic counterpart: it can lead to exponentially faster programs, and also programs that do exponentially less work.

Although we have used results from circuit complexity, we have not "given anything back." It would be interesting to find out if the connection between DPCF programs and monotone circuits has some wider applicability in the area of circuit complexity. It seems unrealistic to hope that it would be easier to prove, say, strong lower bounds for the complexity of slice functions [87] by examining DPCF programs with undefined inputs. But perhaps the more general connection between functional parallel programs with undefined inputs and boolean circuits can be fruitfully exploited.

Chapter 5

Type Inference

This chapter marks the beginning of the second part of the thesis. After our explorations of circuit semantics in the context of parallel extensions of PCF, we return to CDS0 and design a type inference system for it. Our ultimate intent is to use this type inference system to analyze PCF-like languages, taking advantage of the intensional information provided by sequential algorithms. We achieve this goal in the next chapter, where we introduce a high-level lazy, functional language, show how to translate it to CDS0, and build a refinement type inference system on top of our type inference system.

We first present some general considerations in designing a type system for CDS0 in Section 5.1. Section 5.2 discusses CDS0 type definitions in detail. We define the meaning of subtyping and intersection types for ground dcds in Section 5.3, and do the same for sequential algorithms in Section 5.4. A decision procedure for subtyping in the monomorphic case is shown in Section 5.5. We present the monomorphic type inference rules and prove soundness of monomorphic type inference in Section 5.6. Next, we show how polymorphism and overloading can arise in CDS0 in Section 5.7, and describe how to decide subtyping in the presence of type variables in Section 5.8. Finally, Section 5.9 gives the rules for type inference incorporating polymorphism and overloading and shows soundness of the extended system.

5.1 Issues in designing a type system for CDS0

The types in CDS0 are the dcds's. The original intent [5, 7] was that CDS0 would be typechecked. The meaning of typing judgments was taken to be, $x : \tau$ if $x \in D(\tau)$. So the user would type in an expression and a type, and the system would check if the expression belongs to the set of states of that type. Unfortunately, this was never implemented.

Our intent is to devise a type inference system. The motivation for this is twofold: First, CDS0 is, in a sense, a low-level programming language. Writing programs in CDS0 is sometimes difficult, and a type inference system would greatly ease the task. Second, and more important, sequential algorithms form an intensional semantics for sequential programming languages, and we would like a way of extracting the intensional information present in an algorithm. We can imagine sequential algorithms as an intermediate language in the compilation of a functional language, an intermediate language which makes many kinds of analyses of program properties easier. Our type inference system will be the foundation on which we build our program analysis.

There are several implications of our desire to build a type inference system for CDS0, among them the need for a definition of subtyping for dcds's, and the need to introduce intersection types.

We discuss each in turn, also pointing out peculiarities of CDS0 which must be reflected in the type system.

If we take the original meaning of typing judgments and add subtyping, we are forced to say that $\sigma \leq \tau$ if and only if $D(\sigma) \subseteq D(\tau)$, *i.e.,* we get a system in which subtyping is equivalent to inclusion of sets of states. So a supertype will be "bigger" than a subtype: it will have more cells, more values per cell, and a "weaker" (more permissive) enabling relation. It turns out that this notion of subtyping does not accord with the usual notion of subtyping from object-oriented languages [43]. Consider the following definitions of dcds's for points and colored points:

```
let point = dcds
  cell X values [..]
  cell Y values [..]
end;

let cPoint = dcds
  cell X values [..]
  cell Y values [..]
  cell C values red, green, blue
end;
```

With the view of subtyping as inclusion of sets of states it would be the case that $point \leq cPoint$. This is the exact opposite from what would happen in an object-oriented language. For example, the same types defined as record types look as follows (the language is a generic record language, similar to one from [42]):

```
type point = { X : Int, Y : Int };
type color = red | green | blue;
type cPoint = { X : Int, Y : Int, C : color };
```

For record types, a subtype is "more specific" than a supertype, that is, when viewed as a property, it is applicable to fewer records, and so $cPoint \leq point$.

Another problem with the notion of subtyping as inclusion of sets of states is caused by the computation model of CDS0. The model is one of incrementally growing a state, by filling an accessible cell with a value. One would expect that, as we add information, the type would decrease, because we are becoming more specific. However, the exact opposite would happen: given the state $\{X = 3, Y = 4\} : point$, if we add the event $C = red$, we get a *cPoint*, which is higher in the type hierarchy.

For the reasons mentioned above, we want a notion of subtyping for dcds's which makes them more like records. The difficulty in doing this comes from the fact that we cannot offer the same guarantees. A record with type *cPoint* is guaranteed to contain *all* the fields X, Y, and C filled with some value. That way we can "coerce" it to be a *point* by throwing away the color field. In CDS0, with its computation model of incrementally growing a state, we cannot offer the same guarantee because we have to be able to type "incomplete" information. For example, we have to say that $\{C = red\} : cPoint$, even though it does not have the X and Y cells filled.

Our solution is to imagine that each cell in a dcds definition has a variant as a value, one of whose elements is the special value Ω, and the rest of which is the regular list of values from the definition. Recall that according to CDS0's operational semantics (see Appendix A.1), asking for the value of a cell that is not filled in a state produces the result Ω, which is printed by the interpreter as a blank:

```
# {C=red};
request? X;
-->
request? Y;
-->
```

If we take this view, then we can make *cPoint* a subtype of *point* because we can coerce any state of *cPoint* to one of *point* by throwing away events involving cell C. Cells X, Y, might be filled with the special value Ω, but they are guaranteed to be there.

In general, when we assign a type to a state, we will imagine all initial cells present in the type but not filled in the state, filled with Ω. In addition, we want to do the least amount of filling in necessary. For instance, if we are presented with the state $\{X = 3\}$, we will assign it the type *point*, because we have to fill only one cell with Ω, rather than *cPoint*, when we would have to fill two cells. The formalization of these ideas does not take the form presented above (we will not be translating CDS0's type definitions into a type language with variants and the value Ω) but this is the intuition behind the definitions in this chapter.

Given our view of dcds's as records, we can easily encode records in our language. Fields are cells, and the type of a field becomes an enumeration of constituent values. In fact, dcds are much richer, since they have accessibility conditions, but we shall not be using accessibility conditions in any essential way in our definition of subtyping.

Aside from the need for subtyping, another implication of our decision to have type inference is the need for intersection types. All dcds's are user defined, and there is no restriction on different dcds's having distinct cell names and values, so it is possible to have a state belonging to several of them.

If we want to be able to perform type inference on the full version of CDS0, we also need to have polymorphism and overloading. As we have seen in Section 2.3.4, we can write algorithms with generic (*i.e.,* variable) cell names and values. A fully generic algorithm (such as the already presented identity algorithm) will give rise to a polymorphic type, while a partially generic algorithm (cell names and/or values are partially specified) will give rise to an overloaded type.

5.2 The language of types

We now cover dcds definitions in more detail. There are two kinds of dcds's: ground and higher-order. The ground dcds's are all user-defined. The following simplified grammar describes their syntax (the full grammar can be found in Appendix C.1):

$$
\begin{array}{rcl}
\langle ground \rangle & ::= & \langle dcds\_decla \rangle \mid local\ \langle dcds\_decla \rangle\ in\ \langle dcds\_decla \rangle\ end \\
\langle dcds\_decla \rangle & ::= & letrec\ \langle dcds \rangle \mid let\ \langle dcds \rangle \\
\langle dcds \rangle & ::= & \langle ID \rangle\ =\ dcds\ \{\langle component \rangle\}^*\ end \\
\langle component \rangle & ::= & cell\ \langle name \rangle\ values\ \langle value\_list \rangle\ \langle access \rangle \\
 & & \mid graft\ \langle dcds \rangle.\langle name \rangle\ \langle access \rangle \\
\langle value\_list \rangle & ::= & \{\langle name \rangle\}^+ \\
\langle access \rangle & ::= & \epsilon \mid access\ \langle enabling \rangle \mid \langle access \rangle\ or\ \langle enabling \rangle \\
\langle enabling \rangle & ::= & \langle event \rangle \mid \langle event \rangle\ ,\ \langle enabling \rangle \\
\langle event \rangle & ::= & \langle name \rangle\ =\ \langle name \rangle
\end{array}
$$

We have already defined booleans and integers in Section 2.3.1. We provide a few more examples of dcds definitions which will be used in what follows. First we define *refinements* of *bool*; intuitively, a boolean is either true of false:

```
let true = dcds cell B values tt end;
let false = dcds cell B values ff end;
```

Next, we define integer lists. This is another example of the use of recursion and grafting, similar to integer streams from Section 2.3.1:

```
letrec intlist = dcds
  cell EMPTY values true, false
  graft (int.l) access EMPTY = false
  graft (intlist.l) access EMPTY=false
end;
```

The first few cells of *intlist*, together with their access conditions, look like this:

```
# show more 3 intlist;
{
EMPTY values true, false,
(EMPTY.l) values true, false access EMPTY=false,
((EMPTY.l).l) values true, false access (EMPTY.l)=false,
(N.l) values [..] access EMPTY=false,
((N.l).l) values [..] access (EMPTY.l)=false,
(((N.l).l).l) values [..] access ((EMPTY.l).l)=false}
```

Structurally, *intlist* is quite similar to the lazy natural numbers. There is only one initial cell, $EMPTY$. If $EMPTY = false$, we can fill the value of the first integer in the list, $N.l$, and we are allowed to fill $EMPTY.l$. The cells of the form $EMPTY.l\ldots$ form the "backbone" on which the actual integers of the list attach to, like vertebrae.

Following the example in the introductory chapter, we refine *intlist* into empty lists, lists of a single element, and lists of two or more elements. Specifying empty and singleton lists is simple:

```
let empty_intlist = dcds
  cell EMPTY values true
end;

let one_intlist = dcds
  cell EMPTY values false
  cell (N.l) values [..] access EMPTY = false
  cell (EMPTY.l) values true access EMPTY = false
end;
```

Defining lists of two or more elements is complicated by the fact that we need an intermediate dcds definition, yet we do not want the intermediate definition to appear ultimately in our subtype hierarchy and be used for type inference, so we use a *local* definition:

```
local letrec partial_intlist = dcds
  cell (EMPTY.l) values true, false access EMPTY = false
  cell (N.l) values [..] access EMPTY = false
  graft (partial_intlist.l) access EMPTY = false
  end
in let many_intlist = dcds
  cell EMPTY values false
  cell (N.l) values [..] access EMPTY = false
  cell (EMPTY.l) values false access EMPTY = false
  cell ((N.l).l) values [..] access (EMPTY.l) = false
  graft (partial_intlist.l)
  end
end;
```

This definition does what we want; the first few cells look as follows:

```
# show more 6 many_intlist;
{
EMPTY values false,
(N.l) values [..] access EMPTY=false,
(EMPTY.l) values false access EMPTY=false,
((N.l).l) values [..] access (EMPTY.l)=false,
((EMPTY.l).l) values true, false access (EMPTY.l)=false,
(((EMPTY.l).l).l) values true, false access ((EMPTY.l).l)=false,
((N.l).l) values [..] access (EMPTY.l)=false,
(((N.l).l).l) values [..] access ((EMPTY.l).l)=false}
```

The language of our types is given by the following grammar, where g stands for the names of ground dcds definitions:

$$\tau ::= \; g \mid \alpha \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \bigwedge[\tau_1..\tau_n] \mid \{\tau_1..\tau_n\} \mid \forall\alpha.\ \tau$$

We shall use the variables $\tau, \sigma, \delta, \zeta, \xi, \ldots$, to range over types, and α, β, $\ldots$, to range over type variables. We use the notation $\bigwedge[\tau_1..\tau_n]$ for intersection types and $\{\tau_1..\tau_n\}$ for overloaded types; sometimes the range of the subscript will be specified using a "comprehension" notation, $\bigwedge[\sigma_i \mid \ldots]$, and $\{\tau_i \mid \ldots\}$.

5.3 Ground dcds

In what follows it will be convenient to assume that a dcds M is given by a tuple $(C, V_c, \vdash_c)$, that is, a set C of cell names, a family V_c of sets of values for each cell $c \in C$, and a family of enabling (accessibility) relations $\vdash_c$ for each cell. This is slightly different from, but obviously equivalent to, Definition 2.2.1.

5.3.1 Subtyping

Intuitively, there are two cases in which a dcds should be considered a *subtype* of another: As in the example of points and colored points, a subtype could be an *extension* of the supertype, adding more cells. Also, we would expect the refinements of *bool* and *intlist* to be subtypes; in this case, the subtype has fewer cells, and belongs, in a sense, to a *partition* of the supertype. Our definition of subtyping formalizes this intuition.

Since all types are user-defined, the subtyping relation and the types we assign terms will vary depending on what types have been defined. Our definitions will be predicated by a set of defined types, $\mathcal{L}$.

Definition 5.3.1 (Maximal state) *A state x is maximal if $A(x) = \emptyset$.*

Definition 5.3.2 (Subtyping for ground types) *Given the set of defined types, $\mathcal{L}$, and two ground dcds, $\sigma = (C^\sigma, V_c^\sigma, \vdash_c^\sigma)$ and $\tau = (C^\tau, V_c^\tau, \vdash_c^\tau)$, we say that $\sigma \leq \tau$ if either of the following hold:*

1. *σ "extends" τ (written $\sigma <_e \tau$),* i.e.,
   - *(a)* $C^\tau \subset C^\sigma$
   - *(b)* $\forall c \in C^\tau.\ V_c^\sigma \subseteq V_c^\tau$
   - *(c)* $\forall c \in C^\tau.\ \vdash_c^\tau = \vdash_c^\sigma$
2. *σ belongs to a "partition" of τ (written $\sigma \leq_p \tau$),* i.e.,
   - *(a)* $C^\sigma \subseteq C^\tau$
   - *(b)* $\forall c \in C^\sigma.\ V_c^\sigma \subseteq V_c^\tau$
   - *(c)* $\forall c \in C^\sigma.\ \vdash_c^\tau = \vdash_c^\sigma$
   - *(d)* $x \in D(\sigma) \Longrightarrow x \in D(\tau)$ *and any maximal state* $x \in D(\sigma)$ *is also maximal in* $D(\tau)$ *(*i.e., $\not\exists y \in D(\sigma), y \supset x \Longrightarrow \not\exists y \in D(\tau), y \supset x$ *for any* $x \in D(\sigma)$*).*
3. *There exists a finite chain $\zeta_1, \ldots, \zeta_n$ in $\mathcal{L}$ such that $\sigma \leq_* \zeta_1 \leq_* \cdots \leq_* \zeta_n \leq_* \tau$, where $\leq_*$ ranges over $\{<_e, \leq_p\}$.*

Example 5.3.3 *Given our dcds definitions of the previous section, we have the following subtype relation:*

$cPoint <_e point$	$empty\_intlist \leq_p intlist$
$true \leq_p bool$	$one\_intlist \leq_p intlist$
$false \leq_p bool$	$many\_intlist \leq_p intlist.$

It is clear why cPoint is a subtype by extension of point: it has more cells, as many values and the same accessibility condition for the common cells. For the subtyping by partition, the only interesting case is 2d. As can be verified, all maximal states in the subtypes are maximal in the supertypes.

Example 5.3.4 *We provide an example of subtyping in which case 3 of the definition comes into play. Consider the following dcds:*

```
let empty_colored = dcds
  cell EMPTY values true
  cell C values red, green, blue
end;
```

We have empty_colored $<_e$ empty_intlist and empty_intlist $\leq_p$ intlist. However, empty_colored and intlist are incomparable given only the first two cases of the definition. Using case 3 we can conclude that empty_colored $\leq$ intlist.

The definition of subtyping can be a little confusing, because a ground dcds with more states can be either above or below one with fewer states. This might seem to imply that we can have a contradictory situation, where two non-identical dcds's are simultaneously above and below each other. The following propositions show that not to be the case.

Proposition 5.3.5 *It is not possible to have simultaneously $\sigma <_e \tau$ and $\tau \leq_p \sigma$, when $\sigma \neq \tau$.*

Proof: By the definition of subtyping, in order for both $\sigma <_e \tau$ and $\tau \leq_p \sigma$ to hold, the following conditions must hold:

1. $C^\tau \subset C^\sigma$
2. $\forall c \in C^\tau.\ V_c^\sigma = V_c^\tau$
3. $\forall c \in C^\tau.\ \vdash_c^\tau = \vdash_c^\sigma$
4. $D(\tau) \subseteq D(\sigma)$
5. all maximal states of τ are maximal in σ

It is the last condition that cannot be met. There are three possible ways of adding more cells to τ to get σ: (i) the extra cells are not initial, (ii) they are initial, or (iii) there is a combination of initial and non-initial cells. The last two cases clearly cannot work, because we can easily extend any maximal state in τ by filling one of the initial cells. For case (i), it would have to be the case that the extra cells in σ are enabled by states of τ which are not maximal, AND the maximal states of τ contain no enabling substate for cells of σ. But that implies that the states that enable a σ cell cannot be extended in τ to become maximal, *i.e.,* they are already maximal. This is contradictory. $\square$

It might also seem strange to build transitivity into the definition of subtyping, as we have done. The reason for this is that defining subtyping between dcds's with incomparable sets of cells (*i.e.,* given σ, τ such that neither $C^\sigma \subseteq C^\tau$ nor $C^\tau \subseteq C^\sigma$) is difficult to do without making the relation trivial. The way we chose to define subtyping makes it dependent on the set $\mathcal{L}$ of dcds's defined in the system. This implies that it is possible for two dcds's to be unrelated, but to become so with the addition of a new dcds definition (*cf.* Example 5.3.4). We now show that subtyping is a partial order, listing first two obvious lemmas about the properties of $<_e$ and $\leq_p$.

Lemma 5.3.6 *$<_e$ is transitive and (vacuously) anti-symmetric.*

Lemma 5.3.7 *$\leq_p$ is a partial order.*

Proposition 5.3.8 *$\leq$ is a partial order.*

Proof: Reflexivity is obvious from Lemma 5.3.7. Transitivity is built into the definition (case 3). Anti-symmetry is somewhat more complicated to establish. Suppose we have

$$\sigma \leq_* \zeta_1 \leq_* \cdots \leq_* \zeta_n \leq_* \tau, \text{ and}$$
$$\tau \leq_* \xi_1 \leq_* \cdots \leq_* \xi_m \leq_* \sigma,$$

where each $\leq_*$ ranges over $\{<_e, \leq_p\}$. Then it is the case that $C^\sigma \bigcap C^\tau \neq \emptyset$. Departing from σ, for instance, it is impossible to lose all of C^σ through a chain of subtypes, since the accessibility conditions do not change, therefore all initial cells must exist in the subtypes. By cases 1c and 2c of the definition, it follows that $\forall c \in C^\sigma \bigcap C^\tau .\ V_c^\sigma = V_c^\tau$ and $\vdash_c^\tau = \vdash_c^\sigma$. We will now show this is not possible assuming $\sigma \neq \tau$.

When subtyping by extension, we can introduce new dependencies, but only among the newly introduced cells. When subtyping by partition, the only way in which we can "eliminate" cells from the supertype is if they depend on some initial cell, which also appears in the subtype, but with fewer values (not the ones enabling the missing cells; *cf.* part 2d of the definition). The upshot of this is that starting with σ, say, we can eliminate some of C^σ by subtyping, only by reducing the value set of at least one cell that is left over. This will not work because of our earlier requirement of matching values for the surviving piece. On the other hand, we can simply extend σ, thus making $C^\sigma \subset C^\tau$. But then when we try to go the other way (by subtyping from τ to σ), we have to eliminate $C^\tau \setminus C^\sigma$, which cannot be done except by losing values in C^σ as we argued earlier. Therefore, it must be the case that $\sigma = \tau$. □

To summarize, the following rules about subtyping hold (all typing rules are also listed in Appendix A.2 for ease of reference):

(SUB-REFL) $$\sigma \leq \sigma$$

(SUB-TRANS) $$\frac{\sigma \leq \tau \quad \tau \leq \delta}{\sigma \leq \delta}$$

To this we add a rule about subtyping for products, which cannot be derived from our definition of subtyping for ground dcds.

(SUB-PROD) $$\frac{\sigma_1 \leq \tau_1 \quad \sigma_2 \leq \tau_2}{\sigma_1 \times \sigma_2 \leq \tau_1 \times \tau_2}$$

We now formalize the notion of coercion. Given $\sigma \leq \tau$, we would expect there to be a way of uniformly transforming any state of σ into one of τ. In particular, it should be the case that not all states of σ get mapped into the empty state, because, otherwise, any type can be coerced into any type (since the empty state belongs to every type). So we want our coercions to satisfy this extra condition. We will use φ, ψ to range over coercions.

Definition 5.3.9 (Coercion) *A coercion from σ to τ, where σ, τ are two types, is a function $\varphi : \sigma \to \tau$, which is a restriction on the range of the identity on σ, id_σ,* i.e., range $\varphi \subseteq$ range id_σ, *and such that it does not map all states of σ to the empty state,* i.e., range $\varphi \neq \emptyset$, *unless σ is the empty dcds.*

Proposition 5.3.10 *[Subtyping and ground coercions] If $\sigma \leq \tau$, then there exists a coercion $\varphi : \sigma \to \tau$ such that if $x \in D(\sigma)$ then $\varphi(x) \in D(\tau)$.*

Proof: By cases depending on the kind of subtyping.

1. $\sigma <_e \tau$. Then $\varphi(x) = \{c = v \in x \mid c \in C^\tau\}$.

2. $\sigma \leq_p \tau$. Then φ is the identity on σ.

3. There exists a finite chain $\zeta_1, \ldots, \zeta_n$ in $\mathcal{L}$ such that $\sigma \leq_* \zeta_1 \leq_* \cdots \leq_* \zeta_n \leq_* \tau$, where each $\leq_*$ ranges over $\{<_e, \leq_p\}$. Then construct a coercion φ_i for each piece of the chain and $\varphi \equiv \varphi_{n+1} \circ \cdots \circ \varphi_1$.

The first two cases obviously give rise to coercions. For the last case, recall from the proof of Proposition 5.3.8 that $C^\sigma \bigcap C^\tau \neq \emptyset$, therefore the composition of the coercions will also satisfy the coercion requirements. $\Box$

We formalize what we mean by a subtype hierarchy. We will use $\mathcal{S}$ to range over subtype hierarchies. Our definitions will now be predicated by $\mathcal{S}$.

Definition 5.3.11 (Subtype hierarchy) *A subtype hierarchy is a partially ordered set $\langle \mathcal{L}, \leq \rangle$, where $\mathcal{L}$ is the set of defined types, and $\leq$ is the subtype relation.*

We are finally in a position to begin formalizing some of our intuitions from Section 5.1. We define a notion of minimal type with fewest assumptions: If, given a state x, it is the case that $x \in D(\sigma)$ for several σ, we want to call the lowest such σ the minimal type of x (let us defer, for now, the possibility that there is no single lowest σ to the next section, where we introduce intersection types). However, if $x \in D(\sigma)$ and $x \in D(\tau)$ with $\sigma <_e \tau$, then x would require more assumptions to be considered a member of σ, and so we would choose τ as the minimal type with fewest assumptions. Note that we reason about assumptions in a very indirect fashion, as compared to the presentation of Section 5.1.

Definition 5.3.12 (Minimal type with fewest assumptions) *Given a subtype hierarchy $\mathcal{S}$ and a state x, we say that τ is the minimal type with fewest assumptions of x, if $x \in D(\tau)$ and $\nexists \zeta.\ \tau <_e \zeta$ and $x \in D(\zeta)$, and τ is the lowest type with this property.*

Example 5.3.13 *Assuming the dcds definitions of the previous section, the minimal type with fewest assumptions of $\{X = 3\}$ is point, and the minimal type with fewest assumptions of $\{C = red\}$ is $cPoint$.*

The refinement types are the subtypes by partition ($\leq_p$). We are particularly interested in the case when enough partitions of a type are defined so that, taken together, they "cover" the type.

Definition 5.3.14 (Complete partition) *A set of types $\{\sigma_1, \ldots, \sigma_n\}$ is called a complete partition of a type τ if $\forall i.\ \sigma_i \leq_p \tau$ and $D(\tau) = \bigcup_i D(\sigma_i)$ and $\forall i \neq j.\ \sigma_i$ and σ_j are incomparable.*

Example 5.3.15 *$\{true, false\}$ is a complete partition of bool, and $\{empty, one, many\}$ is a complete partition of intlist. If, for instance, true were not defined, then the partition of bool would not be complete.*

5.3.2 Intersection types

In the previous section we ignored the possibility that a state might have several, incomparable, minimal types with fewest assumptions. To handle such cases, we introduce intersection types. In general, the type of a state will be an intersection of types.

Definition 5.3.16 (Minimal type) *Given a subtype hierarchy $\mathcal{S}$ and a state x, we say that $\bigwedge[\tau_1..\tau_n]$ is the minimal type of x if the τ_i are all the incomparable minimal types of x with fewest assumptions.*

We now present our replacement to the original notion of typing judgments in CDS0. The idea is to say that $x : \tau$ even in cases when $x \notin D(\tau)$, provided there exists a type σ below τ for which $x : \sigma$ does hold. Then we can coerce x to something that is in $D(\tau)$. So we still use the original notion of belonging to the set of states, but only to get a "foothold" into the subtype hierarchy, and from there we apply the subtyping rules.

(AND-INTRO) $$\frac{x : \sigma_1 \quad \cdots \quad x : \sigma_n}{x : \bigwedge[\sigma_1..\sigma_n]}$$

(AND-ELIM) $$\frac{x : \bigwedge[\sigma_1..\sigma_n]}{x : \sigma_i}$$

(SUB-AND-R) $$\frac{\forall i.\ \sigma \leq \tau_i}{\sigma \leq \bigwedge[\tau_1..\tau_n]}$$

(SUB-AND-L) $$\bigwedge[\sigma_1..\sigma_n] \leq \sigma_i$$

Figure 5.1: Typing and subtyping for intersection types

Definition 5.3.17 (Meaning of typing judgments for ground states) *Given a subtype hierarchy $\mathcal{S}$ and a state x, we say that $x : \tau$, if $\sigma \leq \tau$, where σ is the minimal type of x.*

Example 5.3.18 *The minimal type of $\{C = red\}$ is cPoint and not point, because it is the case that $\{C = red\} \in cPoint$ and there is no type below cPoint for which this holds. On the other hand, $\{C = red\} \notin point$. However, we do have $\{C = red\} : point$, since $cPoint \leq point$, and so we can coerce $\{C = red\}$ into a state of point (in this case it is $\emptyset$).*

Our typing and subtyping rules for intersection types are summarized in Figure 5.1.

In the general setting of intersection types, we can state and prove a property which we would expect to hold of any typing system based on dcds's, *i.e.*, that incremental computation decreases the type of a state. If we extend a ground state by a new event, we want the type of the new state to be at most as high as before.

Theorem 5.3.19 *Incremental computation does not increase the minimal type of a ground state.*

Proof: Suppose we have two states, x, y, such that $x \prec_c y$, and $x : \sigma, y : \tau$, where $\sigma, \tau \in \mathcal{S}$, our subtype hierarchy. We want to show that the minimal type of y is not above the minimal type of x.

By the definition of typing judgments, the minimal types of x, y will have forms $\bigwedge[\sigma_1..\sigma_n]$ and $\bigwedge[\tau_1..\tau_m]$, respectively. We need to show that $\forall \sigma_i\ \exists \tau_j.\ \tau_j \leq \sigma_i$.

Pick a σ_i. Suppose it is incomparable to all the τ_j. But then it should be included among them as part of the minimal type of y. Now suppose there is a τ_j such that $\sigma_i \leq \tau_j$. It has to be the case that $\sigma_i <_e \tau_j$, otherwise σ_i would have been listed as one of the minimal types of y, instead of τ_j. But then, since $x \in \tau_j$, σ_i should not be among the minimal types of x; τ_j should be listed instead.

Therefore, there is a τ_j such that $\tau_j \leq \sigma_i$. $\Box$

5.4 Sequential algorithms

Higher-order dcds's, whose states are the sequential algorithms, are treated somewhat differently from ground dcds's. Intersection types are also present, and the meaning of a typing judgment does not change, however, an analogue of Theorem 5.3.19 does not hold for higher-order states.

5.4.1 Subtyping

As in the case of products, we cannot apply the definition of subtyping for ground dcds's to the higher-order case. Instead, we have to impose an ordering from outside. The subtyping rule for arrow is conventional and for the same reason; we want it to encode the idea of substitutability. Even though sequential algorithms are more than just functions (they contain intensional information), we should be able to collapse intensional substitutability to the extensional one.

$$\text{(SUB-ARROW)} \qquad \frac{\sigma_2 \leq \sigma_1 \qquad \tau_1 \leq \tau_2}{\sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2}$$

If a higher-order type is a subtype of another, it should still be possible to coerce states of the former into states of the latter.

Proposition 5.4.1 *[Subtyping and higher-order coercions] If $\sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2$, then there exists a coercion $\varphi : (\sigma_1 \to \tau_1) \to (\sigma_2 \to \tau_2)$ such that if $x \in D(\sigma_1 \to \tau_1)$ then $\varphi(x) \in D(\sigma_2 \to \tau_2)$.*

Proof: By SUB-ARROW we have $\sigma_2 \leq \sigma_1$ and $\tau_1 \leq \tau_2$ and by Proposition 5.3.10 there exist coercions $\varphi_{in} : \sigma_2 \to \sigma_1$ and $\varphi_{out} : \tau_1 \to \tau_2$. We define an event-by-event version of φ, called φ_e, based on the kinds of events of $\sigma_1 \to \tau_1$:

1. $xc' = valof\ c$, where $c' \in C^{\tau_1}$, $x \in D(\sigma_1)$ and $c \in C^{\sigma_1}$. Then

   $$\varphi_e(xc' = valof\ c) = \begin{cases} \emptyset, \text{ if } \ \varphi_{in}(x) \neq x, \text{ or } \varphi_{in}(\{c = *\}) = \emptyset, \text{ or } \varphi_{out}(\{c' = *\}) = \emptyset \\ \{xc' = valof\ c\}, \ \text{ otherwise} \end{cases}$$

   The notation $c = *$ means fill cell c with any reasonable value; we are interested if a particular coercion elides all events involving c.

2. $xc' = output\ v'$, where $c' \in C^{\tau_1}$, $v' \in V_{c'}^{\tau_1}$, and $x \in D(\sigma_1)$. Then

   $$\varphi_e(xc' = output\ v') = \begin{cases} \emptyset, \text{ if } \ \varphi_{in}(x) \neq x, \text{ or } \varphi_{out}(\{c' = v'\}) = \emptyset \\ \{xc' = output\ v'\}, \ \text{ otherwise} \end{cases}$$

Then $\varphi(x) = \bigcup(map\ \varphi_e\ x)$. $\square$

Example 5.4.2 *Suppose we are given the following state of bool $\to$ cPoint:*

```
{ {}X = valof B, {B=tt}X = output 3, {B=ff}X = output 4,
  {}C = valof B, {B=tt}C = output red }.
```

Since bool $\to$ cPoint $\leq$ true $\to$ point, there should be a coercion that transforms the above state into one of true $\to$ point. The result of applying the coercion generated by Proposition 5.4.1 is:

```
{ {}X = valof B, {B=tt}X = output 3 }.
```

The meaning of typing judgments in the higher-order case is the the same as in the ground case: again we use "belongs to the set of states of" as a foothold, and then use the subtyping rules.

Definition 5.4.3 (Meaning of typing judgments for algorithms) *Given an algorithm a, we say that $a : \sigma \to \tau$ if $a \in D(\sigma' \to \tau')$, where $\sigma' \to \tau' \leq \sigma \to \tau$.*

We note that this definition has an interesting implication, as established by the following proposition.

Proposition 5.4.4 *If $a : \sigma \to \tau$, then given an input $x : \sigma$, $a.x : \tau$.*

Proof: Assume $a \in D(\sigma' \to \tau')$, where $\sigma' \to \tau' \leq \sigma \to \tau$. Given an $x : \sigma$, *i.e.*, $x \in D(\sigma_{in})$ for some $\sigma_{in} \leq \sigma$, we have $\sigma_{in} \leq \sigma \leq \sigma'$. Therefore, we can coerce the x to a σ', so $a.x \in D(\tau')$, which implies $a.x : \tau$. $\square$

5.4.2 Intersection types

In general, an algorithm will have an intersection type. Suppose the type of the inputs for algorithm a is $\bigwedge[\sigma_1..\sigma_n]$, and the outputs have type $\bigwedge[\tau_1..\tau_m]$. What should the type of the algorithm be in that case? We would not want it to be $\bigwedge[\sigma_1..\sigma_n] \to \bigwedge[\tau_1..\tau_m]$, because, as would be apparent by examination of the SUB-ARROW rule, we would not then be able to apply the algorithm to an input of type σ_i. A natural choice, then, would be $\bigwedge[\sigma_i \to \bigwedge[\tau_1..\tau_m] \mid i \in 1..n]$, but this is inconvenient, because of the nested intersections.

We will construct a canonical type for a with all the intersections at the outer level. The type of a will be $\bigwedge[\sigma_i \to \tau_j \mid i \in 1..n,\ j \in 1..m]$. The justification for this is provided by the following subtyping rule which establishes an equivalence between our canonical type and the natural type one would expect for the algorithm:

$$\text{(SUB-AND-DIST)} \qquad \bigwedge[\sigma \to \tau_1 \ .. \ \sigma \to \tau_n] \leq \sigma \to \bigwedge[\tau_1..\tau_n]$$

The equivalence is due to the fact that we can deduce the opposite of SUB-AND-DIST using SUB-AND-L, SUB-ARROW, and SUB-AND-R. By SUB-AND-L we have $\bigwedge[\tau_1..\tau_n] \leq \tau_i$ for each i. By SUB-ARROW it follows that $\sigma \to \bigwedge[\tau_1..\tau_n] \leq \sigma \to \tau_i$ for each i. Therefore, by SUB-AND-R, we conclude that $\sigma \to \bigwedge[\tau_1..\tau_n] \leq \bigwedge[\sigma \to \tau_1 \ .. \ \sigma \to \tau_n]$.

Definition 5.4.5 (Canonical type for algorithms) *Given an algorithm a, with minimal input type $\bigwedge[\sigma_1..\sigma_n]$ and minimal output type $\bigwedge[\tau_1..\tau_m]$, then the canonical type of a will be given by $\bigwedge[\sigma_i \to \tau_j \mid i \in 1..n,\ j \in 1..m]$.*

Our canonical type provides a valid type for an algorithm a. If $a : \bigwedge[\sigma_i \to \tau_j \mid i \in 1..n,\ j \in 1..m]$, then it is the case that $a : \sigma_i \to \tau_j$ for each i and j. In addition, we can apply a to any input $x : \sigma_i$, for any i, and we can conclude that $a.x : \tau_j$, for all j, therefore, $a.x : \bigwedge[\tau_1..\tau_n]$.

5.5 Decidability of monomorphic subtyping

It is not at all obvious, at first glance, how one can decide when a dcds σ is a subtype of another, τ. There are three problematic requirements: checking when the set of cells of σ is a subset of the set of cells of τ, checking that all states of σ are also states of τ, and verifying the maximality requirement for subtyping by partition. Fortunately, dcds's are infinite regular trees [24], so it is possible to decide each of the above properties.

First, note that deciding subtyping is easy when we have dcds's with a finite number of cells. The interesting case occurs with dcds's that have an infinite number of cells. There are two ways in which such dcds's can arise: a dcds definition with a cell name that contains an infinite interval tag, such as $R.[..]$ (*cf.* Appendix C.1), or by recursion.

The first case is easy to handle, since the representation is very compact. We can check if another dcds contains cells of the type $R.[..]$ simply by comparing the endpoints of the intervals. For the second case, we can view a dcds as a tree: all cells are leaves and grafting creates a subtree. A recursive dcds (with possibly other recursive dcds's embedded in it by grafts) is an infinite regular tree. The access conditions on cells pose no problem, as they also have a regular structure.

We shall first present an algorithm for deciding subset-of on dcds states. The same algorithm can be used, with only minor modifications, to decide inclusion of sets of cells. Note that in both subtyping by extension and by partition, the accessibility condition is required to be the same on the common subset of cells of the subtype and supertype. Without much additional effort, however, we can decide subset-of in the general setting of different access conditions. Consequently, we present a definition of "weaker-than" for accessibility conditions. A weaker access condition is more permissive, *i.e.,* easier to satisfy. We view an enabling relation for a cell c ($\vdash_c$) as consisting of a family $\mathcal{E}$ of sets of events which enable c (also written $\mathcal{E} \vdash c$).

Definition 5.5.1 (Weaker-than access condition) *Given two enabling relations on a cell of a dcds, we say that* $\mathcal{E}_1 \vdash c \ \preceq\ \mathcal{E}_2 \vdash c$ *(read* $\vdash_{1_c}$ *is weaker than* $\vdash_{2_c}$*) iff* $\forall E_2 \in \mathcal{E}_2.\ \exists E_1 \in \mathcal{E}_1.\ E_1 \subseteq E_2$.

The basic idea is to prove subset-of by induction on the size of cell names. A graft will always increase the length of a cell name. As we "unroll" an infinite dcds one layer at a time, all cell names will get longer. If a dcds has not been able to generate the cells from another dcds, we need not continue checking past a point determined by the length of those cells.

Definition 5.5.2 (Cell name length) *The length of a cell name c is the number of tags in it.*

In order to define the concept of unrolling a dcds, we have to fix a representation for dcds's. This is probably easiest to accomplish by using the actual Standard ML representation from our implementation:

```
datatype cva = Plain of cell * value list * access list
             | Delay of idcds * ((cell * value list * access list) ->
                                 (cell * value list * access list))

and idcds = Nonrec of cva list
          | Rec of cva list * ((cell * value list * access list) ->
                               (cell * value list * access list)) *
                              ((cell * value list * access list) ->
                               (cell * value list * access list))
```

The internal representation of a dcds (*idcds*) is a list of cell, values, and access conditions (*cva*) packaged as either a non-recursive or recursive dcds. It is not important what the representation for cells, values, and access conditions is for our purposes. In the case of a recursive dcds, we also package two functions: one to generate the initial recursive step, and another for all subsequent steps (the reason for having two functions is not relevant). A *cva* can be either a plain triplet of a cell, a list of values, and access conditions, or, when we graft a recursive dcds into another, a pair of the grafted dcds and a function that applies the graft tag. In particular, note that something packaged as a non-recursive dcds at top-level can actually have recursive dcds's embedded in it.

The depth of a dcds is a measure of how deeply embedded a recursive graft is. In the case of a non-recursive dcds, it is simply the number of *cva* entries, whereas for recursive dcds's, we increment the number of *cva* entries by one. In both cases, we increment the depth by one plus the maximum depth of any recursive graft. We present the actual Standard ML code for the depth function, since it is very simple.

Definition 5.5.3 (Dcds depth) *The depth of a dcds σ is given by the following mutually recursive functions:*

```
fun findDelay [] = 0
  | findDelay ((Delay(d,_))::l) = 1 + max(depth d, findDelay l)
  | findDelay ((Plain(_,_,_))::l) = findDelay l

and depth (Nonrec cvas) = (length cvas) + (findDelay cvas)
  | depth (Rec(cvas,_,_)) = 1 + (length cvas) + (findDelay cvas)
```

Example 5.5.4 *We apply the definition of depth to some of the previously presented dcds's:*

$$depth(bool) = 1$$
$$depth(intlist) = 3$$
$$depth(many\_intlist) = 8$$

Note that the actual number returned by *depth* is not meaningful in and of itself. It counts the number of *cva* elements with the addition of a large "penalty" for embedded recursive grafts. Most importantly, it is designed to work in conjunction with the unrolling of a dcds, which essentially retraces the computation of *depth*, by listing the *cva* elements of the dcds up to some specified limit.

Definition 5.5.5 (Dcds unrolling) *Unrolling a dcds σ d times builds a cva list of σ, such that any embedded recursion at depth d gets to apply its recursive graft at least once. A simplified version of our implementation (it omits the raising of exceptions with error messages in certain cases) is given in Figure 5.2.*

The function *iterate* takes a single *cva* element and the functions which apply the tags from a recursive dcds, and generates the specified number of new *cva* elements. The function *listCva* takes a list of *cva* elements and emits at least the specified number of elements from the list (assuming the list is long enough). In the case of a nonrecursive dcds definition, *listCva* will list exactly the specified number of *cva* elements. For recursive definitions, it will list at least the number specified.

We have already seen some examples of unrolling a dcds at the beginning of this chapter. The command *show more d σ* from our CDS0 interpreter actually performs $unroll(d, \sigma)$. We refer the reader to Section 5.2 for examples of usage of this function on *intlist* and *many_intlist*.

We list some properties of the definitions of unrolling and depth, which will be needed in establishing the correctness of our decision procedures.

Lemma 5.5.6 *Given a dcds σ with $depth(\sigma) = d_\sigma$, the computation of $unroll(d_\sigma, \sigma)$ results in the computation of $unroll(d, \tau)$, where τ is the innermost embedded recursive dcds in σ, and $d \geq d_\tau$, where d_τ is the depth of τ.*

Proof: By examination of the code for *unroll* it is apparent that it simply retraces the computation of *depth*. When presented with a nonrecursive dcds, *unroll* subtracts one from its running total for each plain *cva* element. When it encounters an embedded recursive dcds τ, *unroll* calls *iterate* on it with an argument equal to the remaining count, which will be at least as large as the depth of

```
fun iterate (0,_,_,_) _ = []
  | iterate (i, first, f1, fi) (c,v,a) =
      if (first)     (* first iteration: apply f1 *)
        then if (i = 1) then [(c,v,a)]
             else let val newCva = f1(c,v,a)
                  in (c,v,a)::(newCva::(iterate(i-2,false,f1,fi) newCva))
                  end
      else let val newCva = fi(c,v,a)
           in newCva::(iterate (i-1,false,f1,fi) newCva)
           end

fun listCva (0, _) = []
  | listCva (i, []) = []
  | listCva (i, cva::l) =
      (case cva of
           (Plain(c,v,a)) => [(c,v,a)] @ (listCva(i-1,l))
         | (Delay(Rec(cvaList,f1,fi),f)) =>
                let val recPart = map (iterate(i,true,f1,fi))
                                      (listCva(i,cvaList))
                    val ordered = flatten recPart
                 in (map f ordered) @ (listCva(i-1,l))
                 end)

fun unroll i (Nonrec cvaList) = listCva(i, cvaList)
  | unroll i (Rec(cvaList,f1,fi)) =
      let val recPart = map (iterate(i,true,f1,fi)) (listCva(i,cvaList))
      in flatten recPart
      end
```

Figure 5.2: Definition of unrolling

τ, depending on its exact position in the list of *cva* elements. Suppose τ is last in the list of a dcds definition σ (least favorable position from our point of view); further, suppose there are i plain *cva* elements, and that dcds τ has depth d_τ. Then the depth of σ will be $d_\sigma = i + 2 + d_\tau$. The call $unroll(d_\sigma, \sigma)$ will then lead to the call $unroll(d_\tau + 2, \tau)$, which establishes our result. □

Proposition 5.5.7 *Given a dcds σ with $depth(\sigma) = d_\sigma$, the computation of $unroll(d_\sigma, \sigma)$ results in a cva list such that the innermost embedded recursive dcds in σ gets to apply its recursive graft at least once.*

Proof: According to Lemma 5.5.6, the call $unroll(d_\sigma, \sigma)$ will result in a call $unroll(d, \tau)$ with τ the innermost embedded recursive dcds, and $d \geq d_\tau$, the depth of τ. Then it suffices to examine what happens when unroll is applied to a simple recursive dcds with an argument equal to the depth of that dcds.

Suppose τ is a simple recursive dcds (*i.e.,* it contains no embedded recursive dcds's) and it has depth d_τ. $Unroll(d_\tau, \tau)$ will call *iterate* with argument d_τ for every *cva* element in τ. There must be at least one *cva* element, hence $d_\tau \geq 2$ and therefore, *iterate* will get to apply at least one recursive graft. □

From the previous two results, it is easy to derive the following Corollary, which will prove useful in what follows.

Corollary 5.5.8 *Given a dcds σ with $depth(\sigma) = d_\sigma$, the computation of $unroll(d_\sigma + n, \sigma)$ results in a cva list such that the innermost embedded recursive dcds in σ gets to apply its recursive graft at least $n + 1$ times.*

When given two infinite dcds's σ and τ, and having to decide whether $D(\sigma) \subseteq D(\tau)$, the idea is to unroll each dcds up to some point and check the finite *cva* lists for subset-of. We need to be careful to unroll enough of each dcds in order to make the comparison. Consider the following example:

```
letrec M1 = dcds
  cell (N.l) values [..]
  graft (M1.l)
end;
```

```
letrec M2 = dcds
  cell (N.l) values [..]
  cell ((N.l).l) values [..]
  graft (M2.s)
end;
```

Dcds M_1 has cells of the form $N.l\ldots$ with one or more l tags while M_2 has cells of the form $N.l.s\ldots$ or $N.l.l.s\ldots$ with zero or more s tags. If we only unrolled M_1 and M_2 once and made the comparison, we would conclude that $D(M_1) \subseteq D(M_2)$, which is false. We need to unroll M_1 enough times to allow it to differentiate itself from the nonrecursive part of M_2, and only then can we unroll M_2 enough times to enable it to generate all of M_1's cells, if it can. With this in mind, we present our decision procedure for subset-of.

Algorithm 5.5.9 (Deciding subset-of for dcds states) *Given two dcds's, σ and τ, to determine if $D(\sigma) \subseteq D(\tau)$ do the following:*

1. *Let $d_\sigma = depth(\sigma)$ and $d_\tau = depth(\tau)$.*
2. *Let $L = unroll(d_\tau, \tau)$.*
3. *Let $name_\tau$ = length of longest cell name in L.*

4. *Let* $L_\sigma = unroll(d_\sigma + name_\tau, \sigma)$.

5. *Let* $name_\sigma$ = *length of longest cell name in* L_σ.

6. *Let* $L_\tau = unroll(d_\tau + name_\sigma, \tau)$.

7. *Compare the finite* L_σ, L_τ *for subset-of.*

Example 5.5.10 *We show the algorithm in operation on the two dcds's defined above,* M_1 *and* M_2*, deciding whether* $D(M_1) \subseteq D(M_2)$*. First we find the depth:*

$$depth(M_1) = 2, \text{ and } depth(M_2) = 3.$$

Unrolling M_2 *3 times produces:*

```
# show more 3 M2;
{
(N.l) values [..],
((N.l).s) values [..],
(((N.l).s).s) values [..],
((N.l).l) values [..],
(((N.l).l).s) values [..],
((((N.l).l).s).s) values [..]}
```

Therefore, the length of the longest cell name in the exposed portion of M_2 *is 4. Now we unroll* M_1 *6 times which produces:*

```
# show more 6 M1;
{
(N.l) values [..],
((N.l).l) values [..],
(((N.l).l).l) values [..],
((((N.l).l).l).l) values [..],
(((((N.l).l).l).l).l) values [..],
((((((N.l).l).l).l).l).l) values [..]}
```

The longest cell name here is 6, so we next unroll M_2 *9 times, in order to give it a chance to generate all cell names in this portion of* M_1*:*

```
# show more 9 M2;
{
(N.l) values [..],
((N.l).s) values [..],
(((N.l).s).s) values [..],
((((N.l).s).s).s) values [..],
(((((N.l).s).s).s).s) values [..],
((((((N.l).s).s).s).s).s) values [..],
(((((((N.l).s).s).s).s).s).s) values [..],
((((((((N.l).s).s).s).s).s).s).s) values [..],
(((((((((N.l).s).s).s).s).s).s).s).s) values [..],
```

```
((N.l).l) values [..],
(((N.l).l).s) values [..],
((((N.l).l).s).s) values [..],
(((((N.l).l).s).s).s) values [..],
((((((N.l).l).s).s).s).s) values [..],
(((((((N.l).l).s).s).s).s).s) values [..],
((((((((N.l).l).s).s).s).s).s).s) values [..],
(((((((((N.l).l).s).s).s).s).s).s).s) values [..],
((((((((((N.l).l).s).s).s).s).s).s).s).s) values [..]}
```

Comparing the finite unrollings of M_1 and M_2 is enough to convince us that $D(M_1) \not\subseteq D(M_2)$.

Theorem 5.5.11 *Given two dcds's, σ and τ, Algorithm 5.5.9 will claim $D(\sigma) \subseteq D(\tau)$ iff it is the case that $D(\sigma) \subseteq D(\tau)$.*

Proof: Completeness is relatively simple to establish. Suppose $D(\sigma) \subseteq D(\tau)$. The only issue is whether we have unrolled τ enough times to let it generate all *cva* elements in L_σ. But τ is being unrolled $d_\tau + name_\sigma$ times, which, according to Corollary 5.5.8, will make the deepest embedded recursive dcds in τ apply its tag at least $name_\sigma + 1$ times. But clearly, there is nothing to be gained by unrolling τ any further, because any new cell names we generate will be strictly longer than the cell names in L_σ. Therefore, if $D(\sigma) \subseteq D(\tau)$, all *cva* elements in L_σ will have been generated by that point.

Correctness is more difficult to establish. We need to prove that if according to Algorithm 5.5.9 $D(\sigma) \subseteq D(\tau)$, then that is indeed the case. To do this, we need to show that σ cannot possibly generate a *cva* element not in τ.

For simplicity, let us assume that each recursive dcds in σ applies a different tag. The proof is by contradiction. Let c be the shortest cell name generated by σ which is not in τ. Then c will have the form:

$$c = name + k_\sigma t + r_\sigma n_\sigma t,$$

where *name* is the part of the cell name that does not contain any of the tags applied by recursive steps, $k_\sigma t$ denotes the addition of k_σ tags t which are the same as the ones from the recursive step but already existed in c prior to any unrolling, and $r_\sigma n_\sigma t$ denotes n_σ unrollings, each of which applies $r_\sigma t$ tags at a time. We have used the addition symbol (+) to denote the application of a tag (*i.e.*, $N.l.l = N + 2l$).

First we will show that $n_\sigma > 2$. In step 2 of the algorithm we unroll τ for d_τ times, so $name_\tau \geq 1$, and in step 4 we unroll σ for $d_\sigma + name_\tau$ times. By Corollary 5.5.8, this means that the deepest recursion in σ applies at least $name_\tau + 1 \geq 2$ tags. But any cell in L_σ is in τ, and so our counterexample cell c must have more than 2 tags.

Since c is, by assumption, the shortest counterexample, its immediate precursors are in τ. Let us denote them c', c''. They have the following form:

$$c' = name + k_\sigma t + r_\sigma (n_\sigma - 1)t,$$
$$c'' = name + k_\sigma t + r_\sigma (n_\sigma - 2)t.$$

Since c', c'' are in τ, there must be a way to construct them in τ. Because the first 4 steps of the algorithm unroll σ enough times to differentiate it from the nonrecursive part of τ, c' and c''

must be generated by τ recursively:

$$c' = name + k_\tau t + r_\tau n_\tau t,$$
$$c'' = name + k_\tau t + r_\tau (n_\tau - k)t,$$

where, as before, $k_\tau t$ denotes the tags coming from the nonrecursive part of τ, and $r_\tau t$ is the number of tags placed by one unrolling of τ. We now show that if τ can recursively generate two successive cells of σ, then it can generate any subsequent one.

By taking the difference of c' and c'' on both the σ and τ side, we get the number of tags that are different. We use the notation $c' - c''$ for this:

$$c' - c'' = r_\sigma t = r_\tau kt.$$

But this means that what takes σ one unrolling to accomplish, can be done with k unrollings of τ, and so we can actually express c in τ:

$$c = name + k_\tau t + r_\tau (n_\tau + k)t.$$

But then c is not a counterexample after all and we have established a contradiction. This means that τ can generate all cells in σ, and that, indeed, $D(\sigma) \subseteq D(\tau)$. □

The same kind of argument can carried out in the more general setting of dcds's with recursive components which apply the same tags. The difference is that a cell name like c will have a more complicated general form, since recursive contributions can come from several places.

We make use of Algorithm 5.5.9 in deciding whether all maximal states in the subtype are maximal in the supertype. In particular, we will use the finite lists of *cva* elements L_σ, L_τ.

Algorithm 5.5.12 (Deciding maximality requirement) *Given σ, τ, such that $D(\sigma) \subseteq D(\tau)$ compute L_σ, L_τ as in Algorithm 5.5.9 and check:*

1. *All initial cells in τ are in σ (with $V_c^\sigma \subseteq V_c^\tau$).*
2. *All cells enabled by the common values (from both σ and τ) of the initial cells in τ are in σ, and so on, recursively.*

The algorithm terminates because we are only examining the finite L_σ, L_τ. Completeness is again simple to establish. For correctness we can use an argument similar to the one from the proof of Theorem 5.5.11. As before, we are examining finite pieces of infinite dcds's that already have the relevant structure of the whole.

Example 5.5.13 *We show how to decide the maximality requirement for one_intlist and intlist. Since one_intlist is not recursive, we do not need to unroll intlist as many times as dictated by Algorithm 5.5.9. It is enough to unroll intlist its depth (3) plus the longest cell name in one_intlist (1). The lists we get then are:*

```
# show more 3 one_intlist;
{
EMPTY values false,
(N.1) values [..] access EMPTY=false,
(EMPTY.1) values true access EMPTY=false}

# show more 4 intlist;
```

```
{
EMPTY values true, false,
(EMPTY.l) values true, false access EMPTY=false,
((EMPTY.l).l) values true, false access (EMPTY.l)=false,
(((EMPTY.l).l).l) values true, false access ((EMPTY.l).l)=false,
(N.l) values [..] access EMPTY=false,
((N.l).l) values [..] access (EMPTY.l)=false,
(((N.l).l).l) values [..] access ((EMPTY.l).l)=false,
((((N.l).l).l).l) values [..] access (((EMPTY.l).l).l)=false}
```

There is only one initial cell in intlist, $EMPTY$ and it is also initial in one_intlist with a smaller set of values. The two cells enabled by $\{EMPTY = false\}$ in intlist are $(N.l)$ and $(EMPTY.l)$ and they are also enabled in one_intlist, again with a smaller set of values. Nothing is enabled by $\{EMPTY = false, (N.l) = n, (EMPTY.l) = true\}$ in intlist and there are no more cells in one_intlist, so the maximal states in one_intlist are indeed maximal in intlist as well.

5.6 Monomorphic type inference

Type inference systems which contain subtyping normally incorporate the notion of substitutability using a *subsumption* typing rule:

$$(\textsc{Sub}) \qquad \frac{a : \sigma \quad \sigma \leq \tau}{a : \tau}$$

We will not have this rule explicitly in our system, because it would not be syntax-directed. Instead it will be absorbed into the other rules.

Our type inference system for expressions is shown in Figure 5.3. The form of the rules deserves some explanation. We consider the rule for application. As noted before, in general, an algorithm will have an intersection type. Suppose algorithm $a : \bigwedge[\sigma_1 \rightarrow \tau_1 \ .. \ \sigma_n \rightarrow \tau_n]$. By applying And-Elim we can deduce $a : \sigma_i \rightarrow \tau_i$ for each i. Suppose algorithm $b : \bigwedge[\delta_1..\delta_m]$. Again, by And-Elim, $b : \delta_j$ for each j. But then for each j for which it is the case that there exists an i such that $\delta_j \leq \sigma_i$, we can apply the subsumption rule to either the type of a or b to get an application argument of the right type, so we can obtain $a.b : \tau_i$. We then collect all the τ_i's for which we can do this in an intersection as the final type of the application. The other rules have a similar form. For instance, the rule for fixpoint also incorporates subsumption. If $a : \sigma_i \rightarrow \tau_i$ with $\sigma_i \geq \tau_i$, then by Sub-Arrow $\sigma_i \rightarrow \tau_i \leq \sigma_i \rightarrow \sigma_i$ and by subsumption $a : \sigma_i \rightarrow \sigma_i$. Therefore, $fix(a) : \sigma_i$. Using And-Elim, And-Intro we can do this for each σ_i that matches the condition.

Now we present our type inference algorithm.

Algorithm 5.6.1 (Monomorphic type inference) *Given an expression e, $\mathcal{A}_m(e)$ is defined as follows:*

1. *If e is a ground state, x, we match it to all dcds in the subtype hierarchy, and construct its minimal type, $\bigwedge[\sigma_1..\sigma_n]$.*

2. *If e is an algorithm a, or equivalently a higher-order state x, we collect all input cells and values, x_{in}, and get its minimal type, $\bigwedge[\sigma_1..\sigma_n]$. We do the same for the outputs, x_{out}, and we get the minimal type $\bigwedge[\tau_1..\tau_m]$. The type of the algorithm will be the canonical type:*

$$\bigwedge[\sigma_1 \rightarrow \tau_1 \ .. \ \sigma_1 \rightarrow \tau_m \ .. \ \sigma_n \rightarrow \tau_1 \ .. \ \sigma_n \rightarrow \tau_m]$$

$$(\textsc{App}) \quad \frac{a : \bigwedge[\sigma_i \to \tau_i \mid i \in 1..n] \quad b : \bigwedge[\delta_1..\delta_m]}{a.b : \bigwedge[\tau_i \mid \exists j.\ \delta_j \leq \sigma_i]}$$

$$(\textsc{Comp}) \quad \frac{a : \bigwedge[\tau_i \to \delta_i \mid i \in 1..n] \quad b : \bigwedge[\sigma_j \to \tau'_j \mid j \in 1..m]}{a|b : \bigwedge[\sigma_j \to \delta_i \mid \tau'_j \leq \tau_i]}$$

$$(\textsc{Fix}) \quad \frac{a : \bigwedge[\sigma_i \to \tau_i \mid i \in 1..n]}{fix(a) : \bigwedge[\sigma_i \mid \sigma_i \geq \tau_i]}$$

$$(\textsc{Curry}) \quad \frac{a : \bigwedge[(\sigma_i \times \sigma'_i) \to \tau_i \mid i \in 1..n]}{curry(a) : \bigwedge[\sigma_i \to \sigma'_i \to \tau_i \mid i \in 1..n]}$$

$$(\textsc{Uncurry}) \quad \frac{a : \bigwedge[\sigma_i \to \sigma'_i \to \tau_i \mid i \in 1..n]}{uncurry(a) : \bigwedge[(\sigma_i \times \sigma'_i) \to \tau_i \mid i \in 1..n]}$$

$$(\textsc{Pair}) \quad \frac{a : \bigwedge[\sigma_i \to \tau_i \mid i \in 1..n] \quad b : \bigwedge[\delta_j \to \zeta_j \mid j \in 1..m]}{< a, b > : \bigwedge[\sigma_i \to (\tau_i \times \zeta_j) \mid \delta_j \leq \sigma_i]}$$

$$(\textsc{Prod}) \quad \frac{a : \bigwedge[\tau_1..\tau_n] \quad b : \bigwedge[\tau'_1..\tau'_m]}{(a, b) : \bigwedge[\tau_i \times \tau'_j \mid i \in 1..n,\ j \in 1..m]}$$

Figure 5.3: Monomorphic type inference rules

3. *If e is a combinator expression, we apply the inference rules.*

Theorem 5.6.2 (Soundness of monomorphic type inference) *If $\mathcal{A}_m(e) = \tau$ then $e : \tau$.*

Proof: By induction on the structure of e:

1. $e \equiv x$. If e is a ground state, we assign it its minimal type, $\bigwedge[\sigma_1..\sigma_n]$, so $\tau \equiv \bigwedge[\sigma_1..\sigma_n]$ and indeed $e : \tau$.

2. $e \equiv a$ or $e \equiv x$ higher-order. When e is a an algorithm, or equivalently, a higher-order state, we get the minimal type for the input and output dcds and construct a canonical type with intersections at the outermost level, $\bigwedge[\sigma_i \to \tau_i \mid i \in 1..n]$. It is the case that for each i, $a : \sigma_i \to \tau_i$, because $a \in D(\sigma_i \to \tau_i)$. Then $a : \bigwedge[\sigma_i \to \tau_i \mid i \in 1..n]$.

3. $e \equiv a.b$. By the induction hypothesis, $a : \bigwedge[\sigma_i \to \tau_i \mid i \in 1..n]$ and $b : \bigwedge[\delta_1..\delta_m]$. According to the definition of typing judgments, in the case of a, for each i, $a \in D(\sigma'_i \to \tau'_i)$ with $\sigma'_i \to \tau'_i \leq \sigma \to \tau$, and, in the case of b, for each j, $b \in D(\delta'_j)$, where $\delta'_j \leq \delta_j$. For the i such that there exists a j with $\delta_j \leq \sigma_i$, we have $\delta'_j \leq \delta_j \leq \sigma_i \leq \sigma'_i$. Then there exists a coercion $\varphi_{in} : \delta'_j \to \sigma'_i$. But then it will be the case that $a.b \in D(\tau'_i)$, therefore $a.b : \tau_i$. Doing this for every i that satisfies the condition, we get $a.b : \bigwedge[\tau_i \mid \exists j.\ \delta_j \leq \sigma_i]$.

4. $e \equiv < a, b >$. By the induction hypothesis, we have $a : \bigwedge[\sigma_i \to \tau_i \mid i \in 1..n]$ and also $b : \bigwedge[\delta_j \to \zeta_j \mid j \in 1..m]$. For i such that there exists j with $\delta_j \leq \sigma_i$, we obtain a coercion $\varphi_{in} : \delta_j \to \sigma_i$. Since $a : \sigma_i \to \tau_i$, by the definition of typing judgments, $a \in D(\sigma'_i \to \tau'_i)$, with

$\sigma_i \leq \sigma_i'$ and $\tau_i' \leq \tau_i$. Similarly, $b \in D(\delta_j' \rightarrow \zeta_j')$ with $\delta_j \leq \delta_j'$ and $\zeta_j' \leq \zeta_j$. Then for some $x : \sigma_i$, $< a, b > .x \in D(\tau_i' \times \zeta_j')$, which implies $< a, b > .x : \tau_i \times \zeta_j$. Then $< a, b > : \sigma_i \rightarrow (\tau_i \times \zeta_j)$. Doing this for all j that meet the requirement, yields the desired type.

5. $e \equiv curry(a)$. By the induction hypothesis, $a : \bigwedge[(\sigma_i \times \sigma_i') \rightarrow \tau_i \mid i \in 1..n]$. By AND-ELIM, for each i, $a : (\sigma_i \times \sigma_i') \rightarrow \tau_i$. This means that $a \in D((s \times s') \rightarrow t)$ with $(s \times s') \rightarrow t \leq (\sigma_i \times \sigma_i') \rightarrow \tau_i$. This implies, by SUB-ARROW, that $\sigma_i \times \sigma_i' \leq s \times s'$ and $t \leq \tau_i$. Further, by SUB-PROD, we have $\sigma_i \leq s$ and $\sigma_i' \leq s'$. But then, by SUB-ARROW again, $s \rightarrow s' \rightarrow t \leq \sigma_i \rightarrow \sigma_i' \rightarrow \tau_i$. Since $curry(a) \in D(s \rightarrow s' \rightarrow t)$, this implies that $a : \sigma_i \rightarrow \sigma_i' \rightarrow \tau_i$. Doing this for all i yields the desired type.

6. Composition and fixpoint are similar to application and pair. Uncurry and product are similar to curry.

□

Completeness, in the sense of always deducing the lowest type, is not possible in this system for undecidability reasons. The user may define a certain type hierarchy which requires us to decide, for instance, whether an algorithm terminates, in order to assign it the lowest type.

Example 5.6.3 *We presented a CDS0 algorithm for boolean negation in Figure 2.3. We illustrate our monomorphic type inference algorithm on the expression not.$\{B = tt\}$. When typing not, we collect all input cells and their values, and all output cells and their values. Let us assume that the only dcds's defined in our system are bool and intlist. The inputs are cell B with values tt, ff, and likewise for the outputs. The only matching dcds is bool, so we conclude:*

$$not : bool \rightarrow bool, \ \{B = tt\} : bool.$$

By APP *we then conclude:*

$$not.\{B = tt\} : bool.$$

5.7 Polymorphism and overloading

We have seen in Section 2.3.4 how to write algorithms with generic (*i.e.,* variable) cell names and values. Variable names begin with the special symbol "\$". When both cell and value references in an algorithm are variable, we get a polymorphic type. When only one or the other is, or their shape is constrained in some way, we get an overloaded type. The other way of getting overloaded types is to have a missing cell or value name, so that we only know either the cell name or the value. In that case, as well, we are reduced to matching what we have to the entire subtype hierarchy, which might result in different, incompatible matches.

We have already encountered the polymorphic identity in Section 2.3.4. Figure 5.4 shows some more examples of polymorphic algorithms and an example of an overloaded algorithm. The first projection is almost the same as the polymorphic identity, except it finds its input in the left side of a product. Conditional is an example of a mix of generic and nonvariable cell names and values. The input to *cond* is of the form $(bool \times \alpha) \times \alpha$ (which explains the tags on the cells) and the output has type α. The last example shows overloading. Minus is an algorithm that will work on any dcds which has some initial cells whose values are pairs of an integer and something else. For instance, if we had the following dcds's defined in the system,

```
let fst = algo
  request $C do
    valof ($C.1) is
      $V: output $V
    end
  end
end;

let minus = algo
  request $C do
    valof $C is
      ($V.$W): output (~$V.$W)
    end
  end
end;
```

```
let cond = algo
  request $C do
    valof ((B.1).1) is
      tt: valof (($C.2).1) is
            $V: output $V
          end
      ff: valof ($C.2) is
            $W: output $W
          end
    end
  end
end;
```

Figure 5.4: First projection, conditional, and an overloaded algorithm

```
let fractions =  dcds
  cell R values ([..].[1..])
end;
```

```
let series = dcds
  cell (R.[0..]) values ([..].[1..])
end;
```

minus should work on both, even though they have disjoint sets of cells.

When typing an algorithm with variable cell and value references, we have to decide, first, if it is a polymorphic or an overloaded reference (polymorphic references have no constraints on the shape of the value, excluding product tags, and both cell and value are variable), and second, we need to look for matches between variables used in the input and output.

For example, in the case of the polymorphic identity, we have $\$C, \$V$ in both input and output, so assume the input has type α and the output β, the matching stage will conclude that $\alpha \mapsto \beta$, and the final type will be $\forall\alpha.\ \alpha \rightarrow \alpha$. The first projection is similar: we have $\$C.1, \$V$ in the input and $\$C, \$V$ in the output, which gives rise to the types $\alpha \times \beta$ and γ respectively. The matching stage will conclude that $\alpha \mapsto \gamma$, so the resulting type is $\forall\alpha.\ \alpha \times \beta \rightarrow \alpha$.

Polymorphic types will be handled in the standard way, using instantiation and generalization rules. There will be difficulties, however, with instantiating polymorphic types. In general, we will have to solve a constraint satisfaction problem. This is discussed in more detail in the next section.

For overloaded types we do not want to use "meet" to put together the various branches, because an algorithm with an overloaded type will not *simultaneously* have all the types in the branches. It depends on particular instantiations. Therefore, we introduce new notation.

Definition 5.7.1 (Overloaded types) *An algorithm a has type $\{\sigma_1 \rightarrow \tau_1 \,..\, \sigma_n \rightarrow \tau_n\}$ (abbreviated $\{\sigma_i \rightarrow \tau_i \mid i \in 1..n\}$), when there exist instantiations of the variable cell and value names in a so that $a : \sigma_i \rightarrow \tau_i$ for each i. The instantiations do not have to be the same for each i.*

Note that it is possible to have a ground state that is overloaded, but it does not make computational sense, and so we will disallow it.

We add the following subtyping rules: one that relates different overloaded types, and one that relates intersection and overloaded types.

$$\text{(Sub-Over)} \qquad \{\sigma_i \rightarrow \tau_i \mid i \in 1..n\} \leq \{\delta_j \rightarrow \zeta_j \mid j \in 1..m\} \qquad \forall j.\ \exists i.\ \sigma_i \rightarrow \tau_i \leq \delta_j \rightarrow \zeta_j$$

$$(\textsc{Sub-Meet-Over}) \qquad \bigwedge[\sigma_i \rightarrow \tau_i \mid i \in 1..n] \leq \{\sigma_i \rightarrow \tau_i \mid i \in 1..n\}$$

Note that we do not treat overloaded types like unions. If we had, we would have had a rule of the form $\sigma_i \rightarrow \tau_i \leq \{\sigma_i \rightarrow \tau_i \mid i \in 1..n\}$, but this is wrong from the substitutability point of view.

Given the above definition, and our earlier type declarations, we want to say that:

$$minus : \{fractions \rightarrow fractions, series \rightarrow series\}.$$

When it comes time to apply an overloaded type we have to be careful because we have no control over what is in the various branches. The only thing we know is that the branches all have the same shape. Therefore, it is possible for an input to match several branches. Since the algorithm does not simultaneously have all the types in its branches, we would have then to *union* the output types of the matching branches. Thus we introduce union types.

Definition 5.7.2 (Union types) *Given two types, σ, τ, the union of σ and τ (written $\bigvee[\sigma,\tau]$) is the least upper bound of σ, τ in the subtype hierarchy.*

We do not introduce union types in their full generality. In particular, we will not have unions belong to any reported type. Rather, they will be used internally. When the union of two types does not exist in the subtype hierarchy, this will result in a type error.

5.8 Subtyping with polymorphic types

When adding polymorphism to our subtyping system, it is not enough to introduce rules of the form $\alpha \leq \sigma$ and $\sigma \leq \alpha$. The problem is that there may be several possible instantiations of α, but that only some meet the requirement. Consider applying an algorithm of type $(\alpha \times \alpha) \rightarrow \alpha$ to $true \times false$. By App, it must be the case that $true \times false \leq \alpha \times \alpha$, which by Sub-Prod implies $true \leq \alpha$ and $false \leq \alpha$. If we just instantiate α to, say, $true$, the whole thing fails. So, in general, what we need to do is solve a constraint satisfaction problem. In our example there is a very simple solution, $\alpha \mapsto \mathit{bool}$, so the resulting type is *bool*.

Algorithm 5.8.1 (Subtyping polymorphic types) *Given the need to satisfy $\sigma \leq \tau$ do the following:*

1. *Recursively break down σ and τ using the subtyping rules, until we arrive at subtyping between ground dcds and type variables.*

2. *Collect all constraints on type variables.*

3. *Pick a type variable. Suppose it is α. Collect all constraints on α from the list. They will have the form:*

$$\alpha \leq \tau_1, \ldots, \alpha \leq \tau_n, \text{ and } \alpha \geq \sigma_1, \ldots, \alpha \geq \sigma_m.$$

4. *If any of the constraints have right hand sides involving type variables go back to previous step and pick another α. If all type variables have constraints with other type variables then:*
   - *If all right hand sides of all constraints of all type variables are all type variables, then attempt to unify them.*
   - *If some right hand sides involve non variables, then choose those first for next step.*

5. *Unify all the τ_i and all the σ_j. This may generate new constraints for some type variables. If unification fails we fail.*

6. *Let the new constraints be $\alpha \leq \tau'$ and $\alpha \geq \sigma'$. If it is not the case that $\sigma' \leq \tau'$ we fail. Else, for all γ in the subtype hierarchy between σ' and τ', make the substitution $\alpha \mapsto \gamma$ for α everywhere, and attempt to resolve all other constraints.*

7. *If attempt fails, backtrack and try another γ. If no more γ we fail.*

8. *Do the same for all remaining type constraints.*

The subtyping algorithm works by essentially trying out all possibilities. It terminates because the subtype hierarchy is finite, and because we require the least upper bound and greatest lower bound of a set of types to actually be present in the subtype hierarchy. That is, if we needed to find something satisfying $true \leq \alpha$ and $false \leq \alpha$ and there were no supertype of both $true$ and $false$, we would fail, rather than return the type $\bigvee[true, false]$. If the algorithm produces a substitution, then it is guaranteed to be correct because we terminate successfully only if all constraints are satisfied. Completeness of the algorithm follows from the fact that we are trying out all possibilities from the subtype hierarchy.

Example 5.8.2 *Suppose our constraints are:*

$$\begin{aligned}
&\alpha \geq false\\
&\alpha \leq \beta\\
&\beta \geq true\\
&\beta \leq bool.
\end{aligned}$$

Then step 4 of the algorithm will have us proceed with the constraints on β first. There is nothing to unify in the right hand sides, so we have $true \leq \beta \leq bool$. We make the substitution $\beta \mapsto true$ and attempt to solve for α, which fails in step 6 of the algorithm, because $false \not\leq true$. We backtrack and try the only remaining choice for β, i.e., *$\beta \mapsto bool$. This succeeds, with $\alpha \mapsto bool$ as well.*

5.9 Type inference with polymorphism and overloading

We add rules for generalization and instantiation, and note that we only apply generalization when typing a polymorphic algorithm, and we only apply instantiation when trying to subtype with type variables. We also need to add rules about overloaded types. Figure 5.5 contains the all the new rules.

There are only slight modifications to make our algorithm for monomorphic type inference work in the more general setting.

Algorithm 5.9.1 (Polymorphic type inference) *Given an expression e, $\mathcal{A}_p(e)$ is defined as follows:*

1. *If e is a ground state, x, we match it to all dcds in the subtype hierarchy, and construct its minimal type, $\bigwedge[\sigma_1..\sigma_n]$.*

2. *If e is an algorithm a, or equivalently a higher-order state x, we collect all input cells and values, x_{in}, and all output cells and values, x_{out} and:*

$$(\textsc{Gen}) \qquad \frac{e : \sigma}{e : \forall\alpha.\ \sigma}$$

$$(\textsc{Inst}) \qquad \frac{e : \forall\alpha.\ \sigma}{e : [\tau/\alpha]\sigma}$$

$$(\textsc{App-Over}) \qquad \frac{a : \{\sigma_i \to \tau_i \mid i \in 1..n\} \quad b : \sigma}{a.b : \bigvee[\tau_i \mid \sigma \leq \sigma_i]}$$

$$(\textsc{Comp-Over}) \qquad \frac{a : \{\tau_i \to \delta_i \mid i \in 1..n\} \quad b : \sigma \to \tau'}{a|b : \sigma \to \bigvee[\delta_i \mid \tau' \leq \tau_i]}$$

$$(\textsc{Fix-Over}) \qquad \frac{a : \{\sigma_i \to \tau_i \mid i \in 1..n\}}{fix(a) : \bigvee[\sigma_i \mid \sigma_i \geq \tau_i]}$$

$$(\textsc{Curry-Over}) \qquad \frac{a : \{(\sigma_i \times \sigma_i') \to \tau_i \mid i \in 1..n\}}{curry(a) : \{\sigma_i \to \sigma_i' \to \tau_i \mid i \in 1..n\}}$$

$$(\textsc{Uncurry-Over}) \qquad \frac{a : \{\sigma_i \to \sigma_i' \to \tau_i \mid i \in 1..n\}}{uncurry(a) : \{(\sigma_i \times \sigma_i') \to \tau_i \mid i \in 1..n\}}$$

Figure 5.5: Type inference rules for polymorphism and overloading

(a) *If e is fully polymorphic, figure out matches between input and output cell names and values and generate polymorphic type. Generalize it.*

(b) *If e is overloaded, figure out matches between input and output and generate overloaded type* $\{\sigma_1 \to \tau_1 \ ..\ \sigma_n \to \tau_n\}$.

(c) *If e is monomorphic, find minimal type for input* $\bigwedge[\sigma_1..\sigma_n]$, *and output* $\bigwedge[\tau_1..\tau_m]$ *and generate canonical type,*

$$\bigwedge[\sigma_1 \to \tau_1 \ ..\ \sigma_1 \to \tau_m \ ..\ \sigma_n \to \tau_1 \ ..\ \sigma_n \to \tau_m]$$

3. *If e is a combinator expression, we apply the inference rules.*

Theorem 5.9.2 (Soundness of polymorphic type inference) *If* $\mathcal{A}_p(e) = \tau$ *then* $e : \tau$.

Proof: By induction on the structure of e. We only discuss some of the new rules, as the old ones carry through unchanged.

1. $e \equiv a$ or $e \equiv x$ higher-order. When e is a an algorithm, or equivalently, a higher-order state, there are several possibilities. If e contains generic cell and value references (of the kind $\$c, \$v$), we generate a polymorphic type and apply Gen. Since the references are generic, it is the case that for any instantiation, e will have that type. If e is overloaded, we collect all matching dcds, and generate an overloaded type $\{\sigma_i \to \tau_i \mid i \in 1..n\}$. It is the case that for each i there exists an instantiation of the cells and values in e such that $e : \sigma_i \to \tau_i$.

2. $e \equiv a.b$, with $a : \{\sigma_i \to \tau_i \mid i \in 1..n\}$ and $b : \sigma$. When $\sigma \leq \sigma_i$ it means that there exists one particular instantiation of a, such that $a.b : \tau_i$. By collecting all such i and computing

$\delta = \bigvee[\tau_i]$ (assuming it exists), we are ensuring that regardless of which instantiation is used, $a.b : \delta$.

3. The other cases are handled similarly to application.

□

Example 5.9.3 *We have already explained how to obtain a polymorphic type for the first projection, fst, from Figure 5.4. We now illustrate step 2a of the algorithm by showing how to generate a type for the conditional algorithm, also found in Figure 5.4.*

The input cells and values of cond are $((B.1).1) = tt, ff$, $((\$C.2).1) = \$V$, and $(\$C.2) = \$W$. The output is $\$C = \$V, \$W$. Because of the product tags, the type generated for the input is $(bool \times \alpha) \times \beta$. The output gets type γ. In the matching stage, we observe that the input from the second and third component of the product is fully polymorphic and matches the output, therefore we make the substitutions $\alpha \mapsto \gamma$ and $\beta \mapsto \gamma$. The final type is then $(bool \times \alpha) \times \alpha \rightarrow \alpha$.

Chapter 6

Refinement Type Inference

In this chapter, we bring together several disparate ideas into a novel application of sequential algorithms, and an example of the practical utility of intensional semantics. In the introduction, we asked the question of how to take advantage of the intensional information present in an intensional semantics for the purpose of program analysis. We provide an answer to that question here, for the semantics of sequential algorithms, by using CDS0 to perform refinement type inference for lazy, functional programming languages. The techniques we use to achieve this are completely different from those used by Freeman and Pfenning, and described in Chapter 2; in fact, they are more closely related to the work of Hughes and Ferguson on abstract interpretation using sequential algorithms, also presented in the same chapter.

In our system, the user may choose to refine a type, by defining finitely many refinements of that type. Any type may be refined, and the user need not explicitly state which types refine which type; this is automatically inferred by the system. A type and its refinements can always be distinguished by examining a finite number of cells, which we shall call *relevant cells* from the point of view of refinement type inference. This is due to the fact that only finitely many refinements of any type can be defined.

When presented with a CDS0 algorithm (*i.e.,* not a combinator expression), it turns out to be quite easy to generate a refinement type by examining its definition, and tracing out all possible execution paths, collecting information about which inputs lead to which outputs. This information about the dependence of parts of the output on parts of the input is used to generate the refinement type.

When presented with a CDS0 combinator expression, we first obtain a *regular* type for it using the framework described in the previous chapter. We then use the regular type to generate the appropriate relevant cells and we enter an interactive questions and answers session with the expression, asking for its value at the relevant cells. The result of the questions and answers session is a state, which is a small approximation of the expression. We obtain a refinement type for the state using the same techniques as for a CDS0 algorithm. Note that we do not have refinement type inference rules; instead we perform abstract interpretation on the expression directly.

We begin with some introductory examples of how to obtain refinement types for algorithms in Section 6.1. We then formalize the notion of refinement type in Section 6.2, and present an algorithm for obtaining refinement types for CDS0 algorithms (or states) in Section 6.3. To give our results a wider applicability, we introduce a simple lazy functional language in Section 6.4, which we also call, *par abus de langage*, PCF, and show how to compile it to CDS0. Then we describe how to perform abstract interpretation on CDS0 expressions in Section 6.5. Finally, in Section 6.6, we show how to obtain a refinement type for an expression, by performing abstract interpretation at the relevant cells. We prove soundness for our refinement type inference system.

6.1 Introductory examples

We begin by introducing some algorithms on integer lists. These algorithms will not only be used to demonstrate how to perform refinement type inference, but will also serve as a categorical combinator compilation target for the list functions of our higher-level language. Figure 6.1 lists algorithms for null, head, tail, and cons. Given our earlier dcds definitions for *bool*, *int*, and *intlist*, the algorithms will have the expected regular types:

$$\begin{aligned}&null : intlist \to bool,\\ &hd : intlist \to int,\\ &tl : intlist \to intlist,\\ &cons : (int \times intlist) \to intlist.\end{aligned}$$

These types will be obtained by our type inference system from the previous chapter, simply by collecting all input and output cells and values, and matching them to the dcds's in the subtype hierarchy.

The algorithms for null and head are very simple. $Null$ just needs to check if the only initial cell, $EMPTY$, is filled with $true$ or $false$. Hd first has to find $EMPTY = false$ in order to be allowed to ask for the value of cell $N.l$, which it copies over to the output.

Tl has two kinds of cells in the output: cells of the form $EMPTY.l.l\ldots$, with zero or more l tags, and cells of the form $N.l.l\ldots$, with one or more l tags. Just as in the case of the algorithms on lazy natural numbers from Chapter 3, we use a variable, `$T`, to stand for zero or more tags. The general form of the cell names will then be $EMPTY.\$T$ and $N.\$T.l$. When trying to fill an output cell of the form $EMPTY.\$T$ we need to copy over the contents of the input cell $EMPTY.\$T.l$. Before asking for the value of that cell, we must list in a from construct an input state that enables our output cell. Similarly for the output cells of the form $N.\$T.l$: we first list an input state that enables those output cells in a from construct, then copy over the contents of input cell $N.\$T.l.l$.

Despite its longer length, *cons* is actually simpler than *tl*, with a similar structure. In this case our output cells have the general form $EMPTY.\$T.l.l$ and $N.\$T.l.l$, which does not include the first three cells of *intlist*, so we list them separately, as base cases. The trees which compute the values of $N.l$ and $EMPTY.l$ do not need a from construct, because the cell $EMPTY$ is guaranteed to be filled with $false$ in the output, regardless of the input.

Now we make the simple observation that instead of collecting *all* input cells and their values and *all* output cells and their values and matching that to various dcds's to obtain a type, a process which essentially flattens the structure of a sequential algorithm, we could take advantage of the tree structure by just collecting input and output cell and value information for each *path* through the algorithm. A path is any sequence of *from* and *valof* statements that ends in an *output* statement. We collect this information for our integer list algorithms in Table 6.1.

Before going further, let us assume that aside from the dcds's *bool*, *int*, and *intlist*, we have also defined the previously encountered refinements of *bool* and *intlist*: $true$, $false$, *empty_intlist*, *one_intlist*, and *many_intlist*. The question that arises is: What happens if, for each algorithm, we assign a type to each of its lines in Table 6.1? The answer turns out to be that we would get the refinement type for the algorithm.

Let us denote a refinement typing judgment by the notation " $:_r$ ". If algorithm a has refinement type τ, we will write this $a :_r \tau$. Further, let us abbreviate the names of the refinements of *intlist*

```
let null = algo
  request B do
    valof EMPTY is
      true : output tt
      false : output ff
    end
  end
end;

let hd = algo
  request N do
    valof EMPTY is
      false : valof (N.1) is
                $V : output $V
              end
    end
  end
end;

let tl = algo
  request (EMPTY.$T) do
    from {(EMPTY.$T)=false} do
      valof ((EMPTY.$T).1) is
        $B : output $B
      end
    end
  end
  request ((N.$T).1) do
    from {(EMPTY.$T)=false,
          ((EMPTY.$T).1)=false} do
      valof (((N.$T).1).1) is
        $V : output $V
      end
    end
  end
end;
```

```
let cons = algo
  request EMPTY do
    output false
  end
  request (N.1) do
    valof (N.1) is
      $V : output $V
    end
  end
  request (EMPTY.1) do
    valof (EMPTY.2) is
      $B : output $B
    end
  end
  request (((EMPTY.$T).1).1) do
    from {((EMPTY.$T).2)=false} do
      valof (((EMPTY.$T).1).2) is
        $B : output $B
      end
    end
  end
  request (((N.$T).1).1) do
    from {((EMPTY.$T).2)=false} do
      valof (((N.$T).1).2) is
        $V : output $V
      end
    end
  end
end;
```

Figure 6.1: Algorithms on integer lists

null	
input	output
`EMPTY = true`	`B = tt`
`EMPTY = false`	`B = ff`

hd	
input	output
`EMPTY = false, N.1 = $V`	`N = $V`

tl	
input	output
`EMPTY.$T = false, EMPTY.$T.1 = $B`	`EMPTY.$T = $B`
`EMPTY.$T = false, EMPTY.$T.1.1 = false, N.$T.1.1 = $V`	`N.$T.1 = $V`

cons	
input	output
$\emptyset$	`EMPTY = false`
`N.1 = $V`	`N.1 = $V`
`EMPTY.2 = $B`	`EMPTY.1 = $B`
`EMPTY.$T.2 = false, EMPTY.$T.1.2 = $B`	`EMPTY.$T.1.1 = $B`
`EMPTY.$T.2 = false, N.$T.1.2 = $V`	`N.$T.1.1 = $V`

Table 6.1: Input and output dependence for integer list algorithms

to *empty*, *one*, and *many*. Then, we would expect to be able to deduce the following refinement typing judgements from Table 6.1:

$$null :_r \bigwedge[empty \to true, one \to false, many \to false]$$
$$hd :_r \bigwedge[one \to int, many \to int]$$
$$tl :_r \bigwedge[one \to empty, many \to intlist]$$
$$cons :_r \bigwedge[(int \times empty) \to one, (int \times one) \to many, (int \times many) \to many]$$

In the rest of this section, we shall attempt to make the inference of such refinement types plausible, by looking at how to do it for the algorithms *null*, *hd*, and *tl* in more detail. We defer a detailed discussion of *cons* to Section 6.3, where we introduce our actual algorithm for performing refinement type inference for algorithms.

The procedure is quite simple for *null* and *hd*. Recall from our dcds definitions for *empty*, *one*, and *many* in Section 5.2, that the event $EMPTY = true$ could only come from dcds *empty*. So the first line in the table for *null* generates the refinement type $empty \to true$. The event $EMPTY = false$ could come from either *one* or *many*. So the second line in *null*'s table generates the "raw" type:

$$\bigwedge[one, many] \to false.$$

By distributing the meet over the arrow (just as we did last chapter in constructing the canonical type for an algorithm), and combining the result with the type of the first line, we arrive at *null*'s final refinement type:

$$null :_r \bigwedge[empty \to true, one \to false, many \to false].$$

The only line in the input/output dependence table for *hd* gives us the following raw type:

$$\bigwedge[one, many] \to int.$$

Again, we distribute the meet over the arrow to obtain the refinement type:

$$hd :_r \bigwedge[one \rightarrow int, many \rightarrow int].$$

The procedure is much more interesting in the case of tl, because we end up with meets on both sides of the arrow in the raw type, and we have to introduce a dependency checking phase, which attempts to eliminate the meets.

The first line of the table for tl gives us the following raw type:

$$\bigwedge[one, many] \rightarrow \bigwedge[empty, one, many].$$

We cannot simply distribute the meets in this case, because cell and value variables are involved, and it is quite possible that certain instantiations of those variables will preclude some of the types generated by a naïve distribution of the meets. What we have to do then is to consider each case in which the input comes from one of the types in $\bigwedge[one, many]$, which will give us certain variable instantiations, and then see what happens to the type of the output when we use those instantiations.

In our particular example, this works as follows: The input can be in either *one*, or *many*. Suppose it is in *one*. That means that $\$T$ gets bound to the empty tag, and $\$B \mapsto true$. This implies that the output event is $EMPTY = true$, which can only come from *empty*. So we have "proven" a dependence between input *one* and output *empty*, therefore the type $one \rightarrow empty$ will be one of the members of tl's refinement type.

Now suppose the input comes from *many*. In this case, there are infinitely many possible instantiations of the variables $\$T$ and $\$B$. We cannot check all of them, but we can unroll *many* enough times so that a "relevant" portion is exposed. This will be made precise later. For now, let us suppose we have unrolled it as in Section 5.2. For each cell name and value list in the resultant *cva* list, we match against the events in the input part of the first line of tl's input/output dependence table. Nothing interesting happens until we come to the following line from the *cva* list:

```
((EMPTY.l).l) values true, false access (EMPTY.l)=false
```

When we match this against $EMPTY.\$T.l = \$B$ from the input, we get $\$T \mapsto l$ and $\$B \mapsto true, false$. Applying this substitution to the output variables, we get $EMPTY.l$ with values $true, false$. It turns out that this can only come from *intlist*, and hence we obtain another piece of the refinement type: $many \rightarrow intlist$. In retrospect, this should not prove surprising, since we are not distinguishing between two and three-element lists: When presented with an input from *many*, tl can either return something in *one*, or something in *many*, so we must lose some precision and give the resultant type of *intlist*.

The second line of the table for tl gives us the following raw type:

$$many \rightarrow \bigwedge[one, many].$$

This merely confirms what we already knew: There are instantiations of the variables such that the output is in *one* and others such that the output is in *many*, therefore we must take the union of *one* and *many* as the type of the result. We then obtain the following refinement type for tl:

$$tl :_r \bigwedge[one \rightarrow empty, many \rightarrow intlist].$$

A natural question that arises after reading our informal description of the type inference algorithm is: What happens when one cannot establish a dependence between input and output

(REF-REFL)	$\tau \sqsubseteq \tau$
(REF-SUB)	$\dfrac{\sigma \leq_p^* \tau}{\sigma \sqsubseteq \tau}$
(REF-AND)	$\bigwedge[\sigma_1..\sigma_n] \sqsubseteq \bigwedge[\tau_1..\tau_m] \qquad \forall i\ \exists j.\ \sigma_i \sqsubseteq \tau_j$
(REF-ARROW)	$\dfrac{\sigma_1 \sqsubseteq \sigma_2 \quad \tau_1 \sqsubseteq \tau_2}{\sigma_1 \rightarrow \tau_1 \sqsubseteq \sigma_2 \rightarrow \tau_2}$
(REF-PROD)	$\dfrac{\sigma_1 \sqsubseteq \sigma_2 \quad \tau_1 \sqsubseteq \tau_2}{\sigma_1 \times \tau_1 \sqsubseteq \sigma_2 \times \tau_2}$

Figure 6.2: Definition of refinement

types? The answer is that one can always establish a dependence when variables are involved. This is due to the fact that we are only dealing with CDS0 algorithms or states and not combinator expressions, hence we do not have recursion (we will show how to handle recursion when we discuss refinement type inference for combinator expressions). It will always be possible to determine, for a certain instantiations of variables in the input, what happens to the output. The only time this will not be possible is when the input and output are actually states that belong to the intersection of several dcds's. For example, the event $EMPTY = false$ belongs to both *one* and *many*. In such cases it is correct to distribute the meet.

6.2 Refinement types

Before presenting in more detail the algorithm for assigning a refinement type to a sequential algorithm, we shall be more precise about what we mean by refinement types and refinement typing judgments.

Intuitively, as described in the previous chapter, where we set the foundations for our typing system, a *refinement* σ of a type τ is a subtype by partition, *i.e.,* such that $\sigma \leq_p \tau$. In general, we will allow a chain of subtypes by partition, $\sigma \leq_p^* \tau$. Of particular interest will be the cases when a type is completely partitioned, as we will not be able to get meaningful refinement types otherwise.

For the non-ground types, we present type inference rules for determining when a type is a refinement of another.

Definition 6.2.1 (Refinement) *We say that $\sigma \sqsubseteq \tau$ (σ refines τ) if it can be deduced from the inference rules in Figure 6.2.*

Note that REF-SUB actually implies REF-REFL since it is always the case that $\tau \leq_p \tau$. We list REF-REFL separately for clarity. The rule REF-ARROW may seem a little strange, because it looks like a covariant subtyping rule, but it means something else: it says that both the input and output refinement types should be refinements of the respective regular types. For instance, we want to have $true \rightarrow false \sqsubseteq bool \rightarrow bool$ (think of the case of boolean negation, from the first chapter).

The rule REF-PROD is not surprising. It codifies what we would expect. For example, it should be the case that $true \times false \sqsubseteq bool \times bool$.

The rule Ref-And says that each member of the left hand side meet must be a refinement of some member of the right hand side meet. Most useful to us will be the case when the right hand side is not a meet, in which case the rule says that all types on the left hand side must refine the right hand side. For example,

$$\bigwedge[\mathit{empty} \rightarrow \mathit{true}, \mathit{one} \rightarrow \mathit{false}, \mathit{many} \rightarrow \mathit{false}] \sqsubseteq \mathit{intlist} \rightarrow \mathit{bool}.$$

Also note that the above definition implies that a refinement type has the same shape as the regular type that it refines.

The meaning of a refinement typing judgment for ground states is the same as for regular typing judgments. However, for algorithms things are different, because an algorithm will not belong as a state to a subtype of the refinement type. Rather, the refinement type is the type of just one "slice" of the algorithm. Consider the algorithm *null*. It is the case that $\mathit{null} \in D(\mathit{intlist} \rightarrow \mathit{bool})$ but $\mathit{null} \notin D(\mathit{empty} \rightarrow \mathit{true})$. Instead, in view of the actual origin of the refinement types as types of paths through an algorithm, the meaning of a refinement type should be of the form: If the input has a certain refinement type, then the output has a certain refinement type. Considering *null* again, if an input x has refinement type *empty* then it will be the case that $a.x$ will have refinement type *true*. With this in mind, we present the following definition.

Definition 6.2.2 (Meaning of refinement typing judgments) *Given a sequential algorithm* $a : \bigwedge[\sigma_i \rightarrow \tau_i \mid i \in 1..n]$, *we say that* $a :_r \bigwedge[\delta_j \rightarrow \zeta_j \mid j \in 1..m]$ *if* $\bigwedge[\delta_j \rightarrow \zeta_j \mid j \in 1..m] \sqsubseteq \bigwedge[\sigma_i \rightarrow \tau_i \mid i \in 1..n]$ *and if for any* j, *given* $x :_r \delta_j$, $a.x :_r \zeta_j$.

As a sanity check, we have the following proposition which relates regular typing judgments to refinement typing judgments.

Proposition 6.2.3 *If* $a : \sigma \rightarrow \tau$ *then* $a :_r \sigma \rightarrow \tau$.

Proof: By induction on σ and τ. In the base case, for a ground type, the meaning of $:$ and $:_r$ is identical. Now suppose it is the case that if $x : \sigma$ then $x :_r \sigma$ and similarly for τ. But according to Proposition 5.4.4, given $x : \sigma$, $a.x : \tau$. This establishes our result. □

Adding refinement types to our subtype hierarchy can change the regular types of algorithms in ways we may not want. For example, if we define the three refinements of *intlist*, the regular type of *tl* becomes $\mathit{many} \rightarrow \mathit{intlist}$. This is due to our requirement that *all* input cells and values belong to the same dcds, and similarly for the output. So, whereas before we could apply *tl* to a one-element list, now we cannot. The solution is to specify to the type inference system which types should only be used as refinements. We introduce a special definition for this purpose:

```
refine true, false, empty_intlist, one_intlist, many_intlist;
```

This way, the refinement types will only be used for refinement type inference. Also, this implies that we will have two subtype hierarchies: a regular one, and a refinement one, which is an extension of the regular subtype hierarchy.

Note that, as opposed to the Freeman-Pfenning approach, we are not telling the system which types are the refinements of a particular type, but simply that it should use certain types only for refinement type inference.

6.3 Refinement type inference for algorithms

We are now almost ready to present our refinement type inference algorithm. There is one additional consideration, aside from the issues raised in previous sections, that must be faced, and that is what to do when we have type variables in the refinement type but not in the regular type. A simple example is provided by the left conjunction *land*, defined in Figure 2.3. The input/output dependence information is shown below:

land	
input	output
`B.1 = tt, B.2 = tt`	`B = tt`
`B.1 = tt, B.2 = ff`	`B = ff`
`B.1 = ff`	`B = ff`

The first two lines of the table give the types $true \times true \rightarrow true$ and $true \times false \rightarrow false$, but the third line results in $false \times \alpha \rightarrow false$. We know from the regular type of this algorithm, $bool \times bool \rightarrow bool$ that the type variable α really corresponds to *bool*. The question is: Should we instantiate it to *bool* only, or also to its refinements?

Note that this question is a very different one from the problem of instantiations of polymorphic refinement type variables in Freeman and Pfenning. In our case it is more a matter of how to *report* the type to the user; such types will not be used with refinement type inference rules. In addition, by Proposition 6.2.3, any choice we make would be correct. We simply want the type returned to the user to give an accurate impression of what the algorithm can do.

In our particular example of *land* and, in general, whenever the type variable matches a ground type, we can instantiate it only to the regular type. This is because such type variables can only occur in the input (remember, we are collecting all paths that end in an output statement, so we could not have a type variable in the refinement type output that does not correspond to a type variable as well in the regular type output). So the type we would report for *land*,

$$\bigwedge[true \times true \rightarrow true, true \times false \rightarrow false, false \times bool \rightarrow false],$$

would accurately imply that we can apply *land* to anything below *bool*, in its right input, and still get $false$, as long as the left input is $false$.

However, if the type variable corresponds to a higher-order type from the regular type, we will instantiate it to all possible refinements. For example, if $\alpha \mapsto bool \rightarrow bool$, our instantiation would be:

$$\begin{aligned}\bigwedge[&true \rightarrow true, true \rightarrow false, true \rightarrow bool, false \rightarrow true,\\ &false \rightarrow false, false \rightarrow bool, bool \rightarrow true, bool \rightarrow false, bool \rightarrow bool].\end{aligned}$$

We now present our algorithm for refinement type inference.

Algorithm 6.3.1 (Refinement types for algorithms) *Given* $a : \bigwedge[\sigma_i \rightarrow \tau_i]$, *to obtain the refinement type of* a *do the following:*

1. *Collect dependence information for each path through the algorithm.*

2. *For each path, find the refinement type for the output and for its respective input.*

3. *If there are type variables in the refinement type, attempt to eliminate them by matching against regular type. If type variable matches a ground type, only instantiate it to regular type, else instantiate it to all possible refinements of higher-order type.*

4. *If both of the input and output types in the previous step contain meets then attempt to eliminate them by examining dependence information. If unsuccessful, distribute the meet. If input is higher-order, do not distribute the left hand side meet.*

5. *If the type of any path is not a refinement of the regular type, eliminate it.*

6. *Eliminate types that are refined by other types (*i.e.*, eliminate less specific types), unless the less specific types are produced by the dependence examination step (step 4).*

7. *If the dependence information implied* $\sigma \rightarrow \tau_1$ *and also* $\sigma \rightarrow \tau_2$ *then replace these types with* $\sigma \rightarrow \bigvee[\tau_1, \tau_2]$.

8. *Meet together the types of all remaining paths.*

Each of the steps of the algorithm is executed in succession. There is no need to iterate. However, the algorithm does call itself recursively in step 2. Since the size of the input for each recursive call is strictly decreasing, the algorithm is guaranteed to terminate.

As an example, consider the operation of this algorithm on *cons*, the longest of the integer list algorithms. Table 6.2 show the various paths through *cons* with their respective types after step 2 of the algorithm. In step 3 we instantiate the type variables obtaining (without listing duplicates, here or later):

$$(\mathit{int} \times \mathit{intlist}) \rightarrow \bigwedge[\mathit{one}, \mathit{many}],$$
$$(\mathit{int} \times \bigwedge[\mathit{empty}, \mathit{one}, \mathit{many}]) \rightarrow \bigwedge[\mathit{one}, \mathit{many}],$$
$$(\mathit{int} \times \bigwedge[\mathit{one}, \mathit{many}]) \rightarrow \mathit{many}.$$

In step 4, we attempt to find dependencies and eliminate meets, and can do so in the second line above. We obtain the following types after step 4:

$$\bigwedge[(\mathit{int} \times \mathit{intlist}) \rightarrow \mathit{one}, (\mathit{int} \times \mathit{intlist}) \rightarrow \mathit{many}]$$
$$\bigwedge[(\mathit{int} \times \mathit{empty}) \rightarrow \mathit{one}, (\mathit{int} \times \mathit{one}) \rightarrow \mathit{many}, (\mathit{int} \times \mathit{many}) \rightarrow \mathit{many}]$$
$$\bigwedge[(\mathit{int} \times \mathit{one}) \rightarrow \mathit{many}, (\mathit{int} \times \mathit{many}) \rightarrow \mathit{many}]$$

During step 6 of the algorithm we get rid of the first and third lines, which are both refined by the second one, and we obtain the final refinement type:

$$\mathit{cons} :_r \bigwedge[(\mathit{int} \times \mathit{empty}) \rightarrow \mathit{one}, (\mathit{int} \times \mathit{one}) \rightarrow \mathit{many}, (\mathit{int} \times \mathit{many}) \rightarrow \mathit{many}].$$

We now provide more detail and justification for the various steps of the algorithm. Step 4 is the dependence examination step. We have a type of the form

$$\bigwedge[\sigma_1..\sigma_i] \rightarrow \bigwedge[\tau_1..\tau_j],$$

and we attempt to find matches between the σ_i's and the τ_j's. As described before, we will let the input be in each of the σ_i's and see if that narrows down the choice of τ_j's for the output.

We need to decide how much to unroll each of the σ_i. Suppose the line of the dependence table that gave rise to our type had cells with maximum length l. Further, suppose that the maximum depth of any of the σ_i, τ_j is d. Then we would unroll each of the types $d + l$ times. In view of the results of the previous chapter, this would guarantee that our types would have a chance to generate any appropriate cell in the dependence table.

cons		
input	output	"raw" type
$\emptyset$	`EMPTY = false`	$\alpha \rightarrow \bigwedge[one, many]$
`N.1 = $V`	`N.1 = $V`	$(int \times \alpha) \rightarrow \bigwedge[one, many]$
`EMPTY.2 = $B`	`EMPTY.1 = $B`	$(\alpha \times \bigwedge[empty, one, many]) \rightarrow \bigwedge[one, many]$
`EMPTY.$T.2 = false,` `EMPTY.$T.1.2 = $B`	`EMPTY.$T.1.1 = $B`	$(\alpha \times \bigwedge[one, many]) \rightarrow many$
`EMPTY.$T.2 = false,` `N.$T.1.2 = $V`	`N.$T.1.1 = $V`	$(\alpha \times \bigwedge[one, many]) \rightarrow many$

Table 6.2: First two steps in finding refinement type for *cons*.

Step 5 eliminates some types that may arise due to subtyping by extension. For example, if the regular type of an algorithm were $cPoint \rightarrow bool$, it is possible that a path through the algorithm will have type $point \rightarrow true$. We will not accept this as a refinement, because it does not involve subtyping by partition. There is no meaningful way in which a *point* can be considered a refinement of a *cPoint*.

Step 6 eliminates types that may have resulted from instantiation of type variables, which are, therefore, not as precise as types where we have established dependence between certain input and output types. We have seen an example of this in the operation of the algorithm on *cons*.

Step 7 takes into account the situations when we must lose precision in the refinement type. If it is the case that our dependence information implied both $\sigma \rightarrow \tau_1$ and also $\sigma \rightarrow \tau_2$ then we cannot know which is the resultant type given an input in σ so we must replace these types with $\sigma \rightarrow \bigvee[\tau_1, \tau_2]$. The union $\bigvee[\tau_1, \tau_2]$ is guaranteed to exist, because both τ_1, τ_2 are refinements of the same type (which we ensured in step 5). Step 7 does not apply to cases when we could not establish dependencies in step 4 and distributed a right hand side meet. As described before, such cases arise in the absence of variables in the cell names and values, when a state belongs to several dcds's.

Theorem 6.3.2 *[Soundness of refinement type inference] If, according to Algorithm 6.3.1, $a :_r \tau$ then, indeed, $a :_r \tau$.*

Proof: In broad outline, Algorithm 6.3.1 works by assigning a refinement type to each input and each output in all paths through the algorithm. This certainly leads to sound refinement types, according to our definition of the meaning of refinement typing judgments. The only problems might be caused by the modifications we make to the refinement types along the way, such as eliminating meets, instantiating type variables, and taking unions. We discuss each of these in turn.

We have already discussed the instantiation of type variables from step 3 of the algorithm. Because of Proposition 6.2.3, it is sound to instantiate regular types. Certainly instantiating refinement types also is sound.

The elimination of meets from step 4 is sound because we will either prove dependence or have a state that belongs to several dcds's. As we have argued before, we can always prove dependence when cell and values variables are involved, because we do not have recursion at this level. As can be seen from the CDS0 syntax in Appendix C.1, the constraints which can be placed on variables are very simple, and easily decidable.

Finally, when we take unions in step 7, it still true that given $x :_r \sigma$, $a.x :_r \bigvee[\tau_1, \tau_2]$, because $\tau_1 \leq \bigvee[\tau_1, \tau_2]$, and same for τ_2.

In conclusion, each path will have a refinement type $\sigma \to \tau$ such that, given $x :_r \sigma$, $a.x :_r \tau$. Also, we explicitly eliminate any types which are not refinements of the regular type in step 5. Therefore, our algorithm will lead to a refinement of the original type. □

6.4 A higher-level language

As we have already explained when we introduced CDS0 in Section 2.3, the language was meant to be a compilation target from the beginning. The examples presented so far make it quite clear that one would not want to be programming directly in CDS0, unless one wanted to write programs which take advantage of its intensional features, and which cannot be written in an extensional language. For most purposes, one would want to use a higher-level language. We introduce such a language in this section, and show how to compile it to CDS0.

6.4.1 PCF

The higher-level language is a lazy, higher-order, polymorphic, functional language. We call it PCF for historical reasons: the original CDS0 interpreter of Devin [30] also had a PCF interpreter, which actually corresponded to the original PCF [70]. We started with a similar language but added more features, until we arrived at a full-featured functional language. We have kept the name.

A slightly simplified grammar for the language is given in Figure 6.3. The full version can be found in Appendix C.2. The language is typed in the usual way.

The syntax is somewhat similar to that of Standard ML of New Jersey: the binding of identifiers to expressions, lambda abstraction, products, and list functions are the same. The first and second projections are slightly different, as are some of the basic operations. The main difference comes in the definition of recursive functions, where we have sacrificed some ease of readability for ease of compilation.

The programs for boolean negation and for *map* from the introduction are examples of programs written in this language.

6.4.2 Compilation to CDS0

The compilation to CDS0 is the same as that to categorical combinators [26]; the combinators denote sequential algorithms, therefore, an entire PCF program will also denote a sequential algorithm.

The idea of the compilation is to have variables stored in an environment, and have a PCF expression denote a function from the environment to its value. An environment containing variables $x_0, \ldots, x_n$ is implemented as nested pairs of the form $((\ldots(\{\}, x_n), \ldots), x_0)$, where $\{\}$ denotes the empty environment: in our case the empty CDS0 state.

The first step in the compilation is translation of the PCF expression to de Bruijn notation [26], in which variables are replaced by natural numbers. The de Bruijn terms are built as follows:

1. Any natural number is a term,

2. If M and N are terms, then MN is a term,

3. If M is a term, then $\lambda.\ M$ is a term,

4. If M is a term, then $fix\ M$ is a term, and so on for all other PCF built-in functions.

$$
\begin{array}{rcl}
\langle program \rangle & ::= & \langle expr \rangle \\
 & | & val\ id\ =\ \langle expr \rangle \\
\langle expr \rangle & ::= & \langle const \rangle \\
 & | & id \\
 & | & \langle expr \rangle\ \langle expr \rangle \\
 & | & fn\ id\ =>\ \langle expr \rangle \\
 & | & let\ id\ =\ \langle expr \rangle\ in\ \langle expr \rangle\ end \\
 & | & letrec\ id\ =\ \langle expr \rangle\ in\ \langle expr \rangle\ end \\
 & | & \langle expr \rangle\ \langle op \rangle\ \langle expr \rangle \\
 & | & if\ \langle expr \rangle\ then\ \langle expr \rangle\ else\ \langle expr \rangle \\
 & | & (\langle expr \rangle, \langle expr \rangle) \\
 & | & fst\ (\langle expr \rangle) \\
 & | & snd\ (\langle expr \rangle) \\
 & | & \langle expr \rangle\ ::\ \langle expr \rangle \\
 & | & hd\ \langle expr \rangle \\
 & | & tl\ \langle expr \rangle \\
 & | & [\,] \\
 & | & null\ \langle expr \rangle \\
 & | & (\langle expr \rangle) \\
\langle const \rangle & ::= & true \mid false \mid integers \\
\langle op \rangle & ::= & + \mid - \mid * \mid / \mid = \mid < \mid > \mid <= \mid >= \mid and \mid or
\end{array}
$$

Figure 6.3: Syntax of higher-level language

$$
\begin{array}{l}
x_{DB(x_0,\ldots,x_n)} = \min(j \mid x = x_j) \\
(fn\ x\ \texttt{=>}\ M)_{DB(x_0,\ldots,x_n)} = \lambda.\ M_{DB(x,x_0,\ldots,x_n)} \\
(MN)_{DB(x_0,\ldots,x_n)} = M_{DB(x_0,\ldots,x_n)} N_{DB(x_0,\ldots,x_n)} \\
(let\ x\ =\ M\ in\ N)_{DB(x_0,\ldots,x_n)} = ((fn\ x\ \texttt{=>}\ N)\ M)_{DB(x_0,\ldots,x_n)} \\
(letrec\ x\ =\ M\ in\ N)_{DB(x_0,\ldots,x_n)} = ((fn\ x\ \texttt{=>}\ N)\ (fix\ (fn\ x\ \texttt{=>}\ M)))_{DB(x_0,\ldots,x_n)} \\
(M, N)_{DB(x_0,\ldots,x_n)} = (M_{DB(x_0,\ldots,x_n)}, N_{DB(x_0,\ldots,x_n)}) \\
(fst\ M)_{DB(x_0,\ldots,x_n)} = fst\ M_{DB(x_0,\ldots,x_n)} \\
\ldots
\end{array}
$$

Figure 6.4: Translation to de Bruijn notation

The translation of a term M is called $M_{DB(x_0,\ldots,x_n)}$, where $FV(M) \subseteq \{x_0, \ldots, x_n\}$, and it works by keeping track of the variables already encountered, which act as an environment. It is shown in Figure 6.4. We have listed only one of the constants of PCF; the translation for the others is similar.

Now that we have replaced variables with natural numbers, we will use the numbers as indices into the environment. For instance, variable 0 will become *snd*, variable 1 will be mapped to $snd \mid fst$ and so on. A variable becomes code which pulls out a particular location from the environment.

The translation of a term M in de Bruijn notation to categorical combinators, denoted M_{CC}, is shown in Figure 6.5, again listing only some of the constants from the de Bruijn notation as the code for the others is similar.

The translation deserves some comment. The basic constants of the language, *i.e., true*, $false$, and the integers, are encoded as non-strict algorithms from the environment to states of *bool* or *int*. The algorithm $curry(fst)\ .\{B = tt\}$, for instance, has type $\forall\alpha.\ \alpha \rightarrow bool$. When applied to an environment, it ignores it and returns a state of bool.

Lambda abstraction becomes currying. In the code for application, $uncurry(id)$ is the CDS0 application algorithm. When taking fixpoints, we need a version of the fixpoint combinator which works with environments; we call it Y_{env}. It has the type $\forall\alpha.\ (env \rightarrow \alpha \rightarrow \alpha) \rightarrow env \rightarrow \alpha$, and its implementation is:

```
let Yenv = curry(Y | uncurry(id));
```

Y is the normal fixpoint combinator, of type $\forall\alpha.\ (\alpha \rightarrow \alpha) \rightarrow \alpha$, which we implement by doing a "manual" translation from PCF to CDS0. In PCF the code for Y is:

$$fix\ (fn\ f\ \texttt{=>}\ fn\ x\ \texttt{=>}\ x\ (f\ x)),$$

which becomes the following CDS0 code:

```
let Y = fix((curry(curry(uncurry(id) |
                         <snd, uncurry(id) | <snd|fst, snd>>))).emptyenv);
```

We need to define a CDS0 algorithm for every built-in function of PCF. We have already defined the identity (*id*), first projection (fst), conditional (*cond*), left conjunction *land*, and the

$$
\begin{aligned}
&true_{CC} = curry(fst)\ .\ \{B = tt\}\\
&false_{CC} = curry(fst)\ .\ \{B = ff\}\\
&n_{CC} = curry(fst)\ .\ \{N = n\}\\
&x_{iCC} = snd \mid fst^i\\
&(\lambda.\ M)_{CC} = curry(M_{CC})\\
&(MN)_{CC} = uncurry(id) \mid\ < M_{CC}, N_{CC} >\\
&(M, N)_{CC} =< M_{CC}, N_{CC} >\\
&(fix\ M)_{CC} = Y_{env}\ .\ M_{CC}\\
&(fst\ M)_{CC} = fst \mid M_{CC}\\
&(snd\ M)_{CC} = snd \mid M_{CC}\\
&(if\ M\ then\ N_1\ else\ N_2)_{CC} = cond \mid\ << M_{CC}, N_{1CC} >, N_{2CC} >\\
&(M\ and\ N)_{CC} = land \mid\ < M_{CC}, N_{CC} >\\
&\ldots
\end{aligned}
$$

Figure 6.5: Translation to categorical combinators

list algorithms. We give a sampling of the others, by defining addition and integer equality test. The complete list of CDS0 algorithms which are used to compile PCF is given in Appendix B.3.

The algorithm for addition has type $int \times int \to int$ and it works by using variables to record the values of the inputs and adding them:

```
let plus = algo
    request N do
        valof (N.1) is
            $V1: valof (N.2) is
                     $V2: output $V1 + $V2
                 end
        end
    end
end;
```

The integer equality test, of type $int \times int \to bool$, works in similar fashion (the notation “!=” means “not equal to”):

```
let equal = algo
    request B do
        valof (N.1) is
            $V1: valof (N.2) is
                     $V2 with $V2 = $V1: output tt
                     $V2 with $V2 != $V1: output ff
                 end
        end
    end
end;
```

Given a PCF program M, the final result of the compilation to CDS0 is $M_{CC} . \{\}$, *i.e.,* we apply the categorical combinator translation of the program to the empty environment. For examples of translations of PCF programs, we turn to *not* and *map* from the introduction. We instruct our PCF interpreter to print out the code for the two programs:

```
$ print not;
not = curry (cond | < < snd, curry (fst) . B=ff >,
                      curry (fst) . B=tt >) . emptyenv
$ print map;
map = uncurry (id) | < curry (snd),
      Yenv . curry (curry (curry (cond | < < null | snd, curry (fst) . nil >,
         cons | < uncurry (id) | < snd | fst, hd | snd >,
            uncurry (id) | < uncurry (id) | < snd | fst | fst, snd | fst >,
               tl | snd > > >))) > . emptyenv
```

6.5 Abstract interpretation

When performing refinement type inference on CDS0 expressions or PCF programs, we will need to ask for the values of various cells in the expression. This section describes how we can do that without looping, and without always having to resort to a hard bound on the number of recursive iterations.

6.5.1 Loop detection

The Hughes and Ferguson approach to loop detection for sequential algorithms, which we described in Section 2.5, does not work well in our case because we use the CDS02 operational semantics (which we chose for overall efficiency over CDS01, as explained in Section 2.3.7). Our cell names can incorporate expressions, and grow quite large, therefore we cannot simply check for equality of cell names to detect when a cell depends on itself. It is possible to simplify the cell names, and collect information about dependence of output cells on input cells, as Hughes and Ferguson do. In fact, our first approach had this form, but it did not detect many loops, and, just like the Hughes and Ferguson implementation, was extremely space-inefficient. This forced us to develop an alternative approach, one that is better suited to CDS02.

Recall from Section 2.3.7, that the fixpoint rule has the form:

$$\text{(FIX')} \qquad fix(A)\ ?\ c \rightarrow A.fix(A)\ ?\ c$$

In the process of computing $fix(A)\ ?\ c$, we may ask the questions $fix(A)\ ?\ c_1$, $fix(A)\ ?\ c_2$, and so on. If it is the case that, for some index i, we end up wanting to know $fix(A)\ ?\ c_i$, with $c_i = c$, then, clearly, this is a looping computation.

The problem is that the cell names c_i, c, may contain expressions embedded into them, so we cannot easily check for equality. What we shall do is apply a stock simplification to the cell names, without trying to evaluate the expressions inside, and check for syntactic equality. This works fairly well for PCF programs, and does not usually apply to CDS0 programs.

The simplification we perform is to replace cell names of the form $(snd.(x_1, x_2))c$ with x_2c. It turns out that when recursions get unwound in PCF, expressions of the form $snd.(x_1, x_2)$ get concatenated to cell names. This is due to the actual code for Y from the previous section, which is the only possible source for fixpoint computations in PCF. The questions we ask when evaluating PCF expressions will have the form $Y\ ?\ xc$, where x is a complex expression.

We give an example of how this works in practice. Consider the following PCF program:

```
val loop10 = letrec f = fn b => if b then true else f b in f end;
```

This program clearly loops when presented with a $false$ input. The first two unrollings of the fixpoint computation when the question $loop10\ ?\{B = ff\}B$ is asked, will lead to questions of the form $f\ ?\ c_1$ and $f\ ?\ c_2$ with $Y = fix\ f$, and the cells being:

```
c1 = uncurry (id) . (curry (curry (cond | <<snd, curry (fst) . {B=tt}>,
       uncurry (id) | <snd | fst, snd>>)), emptyenv)
     {B=ff} B

c2 = snd . ((emptyenv, Y), uncurry (id) . (curry (curry (cond |
       <<snd, curry (fst) . {B=tt}>, uncurry (id) | <snd | fst, snd>>)),
       emptyenv))
     snd . ((emptyenv, (uncurry (id) | <snd | fst, snd>) . ((emptyenv, Y),
       uncurry (id) . (curry (curry (cond | <<snd, curry (fst) . {B=tt}>,
       uncurry (id) | <snd | fst, snd>>)), emptyenv))), {B=ff})
     B
```

Applying the simplification to c_2 shows it to be syntactically equal to c_1, and so we have proved that $loop10$ deserves its name.

6.5.2 Depth-bounding

We cannot detect all loops in the manner presented above. Even fairly simple functions, which loop on some input which is slightly modified and then modified back to the original form, cannot always be detected. For example, the following PCF function loops:

```
val loop9 = letrec f = fn l => if null l then let l1 = f (1::l) in f l1 end
                               else f (tl l)
            in f end;
```

This cannot be detected, because we do not simplify expressions of the form $tl\ (x::l)$ to l. We could add such new simplification rules, but we would, of course, still not be able to detect all loops because it is undecidable.

In fact, since our language is lazy, we have to contend with infinite data structures. It is possible to define, for instance, an infinite list of ones:

```
val ones = Y (letrec f = fn l => 1::(f l) in f end);
```

If we had a *length* function, detecting a loop in *length ones* would not be possible, even in the Hughes and Ferguson approach. So rather than add more simplification rules, we add a bound on the number of recursive iterations, a bound already made necessary by the presence of infinite data structures.

We modify the operational semantics of fixpoint to keep track of how many times it has been called while computing a value for the same cell. If that reaches a certain bound, we interrupt the computation. We note that, in practice, the bound can be set to a very low value (for instance, 30), since if an expression will not loop, its computation will unroll to a shallow depth. This is dependent on the dcds's defined in the system, and in particular on the refinements (as will be explained in the next section). The refinements we use can be distinguished by examining at most three cells, which usually leads to short computations.

6.6 Refinement type inference for expressions

Given a CDS0 combinator expression, or equivalently, a PCF program, we shall perform refinement type inference for it by first obtaining its regular type, seeing if the regular type admits any possible refinements, then generating an initial set of relevant cells with which to query our expression. In the process of querying the expression, we will have uncovered a state, which is a small approximation to the combinator expression. We then perform refinement type inference on the state using the techniques of Section 6.3.

A ground type will admit refinements when it has subtypes by partition in the refinement subtype hierarchy. A compound type will *not* admit refinements when either:

- The type is fully polymorphic, or
- No components of the type admit refinements.

For example, if *int* has no refinements, the type $int \rightarrow int$ does not admit refinements, but $int \rightarrow bool$ does, when refinements $true$ and $false$ are defined.

6.6.1 Generating relevant cells

A given type and its refinements can always be distinguished by examination of a finite number of cells. This is due to the fact that only finitely many refinements of a type can be defined. In this section we discuss how to automatically generate such cells. First we consider ground dcds's.

Recall from Definition 5.3.2, that a subtype by partition has the same initial cells as the supertype, but some may have fewer values. We are interested in exactly those cells. In particular, we want those distinguishing cells which do not have an infinite number of possible values. After we collect such initial cells, we shall look at all cells enabled by the initial cells, and collect distinguishing cells, and so on. We stop when there is no difference between the supertype and the subtype's cells.

Algorithm 6.6.1 (Generating ground relevant cells) *Given a type τ and a collection of its refinements, σ_i, the set of relevant cells is generated as follows:*

1. *Find maximum depth among τ, σ_i, and unroll each dcds to that depth.*
2. *Collect all initial cells from τ and the σ_i. Let us call such sets of cells, I_τ, I_{σ_i}.*
3. *Compare the I_{σ_i} among each other: if the same cell exists in two of the I_{σ_i}, but with different values, and the set of values is not infinite, add it to the list of relevant cells.*
4. *Compare the I_τ to the I_{σ_i} as above.*
5. *Generate the set of cells reachable from the initial cells for τ and the σ_i, and perform the same comparison as above. When adding a non-initial relevant cell, add also the cells which may enable it.*

6. *If there is no difference between the sets of cells for two σ_i, or a σ_i and τ at this stage, but there was in the previous generation of cells, add all cells with non-infinite value lists.*

7. *Continue in this fashion, until all cells are the same, or no more cells can become enabled.*

Example 6.6.2 *As an example, we apply Algorithm 6.6.1 to intlist and its refinements, empty, one, and many. The maximum depth among the four dcds's, as we have seen in the previous chapter, is 8 (the depth of many). The sets of initial cells generated from unrolling each of the four dcds's 8 times are:*

$$\begin{aligned}&I_{intlist} = I_{empty} = I_{one} = I_{many} = \{EMPTY\}\\&Relevant = \emptyset\end{aligned}$$

Cell $EMPTY$ exists in all I sets and it does have a different set of values, because of empty. Furthermore, the set of values is not infinite. Therefore we add it to the set of relevant cells. We generate the new I sets:

$$\begin{aligned}&I_{intlist} = I_{one} = I_{many} = \{N.l, EMPTY.l\}\\&Relevant = \{EMPTY\}\end{aligned}$$

Cell $N.l$ has the same set of (infinite) values in each of the I sets, so we discard it. However, $EMPTY.l$ again has different (finite) sets of values, so we add to the relevant cells, and generate the next level of reachable cells:

$$\begin{aligned}&I_{intlist} = I_{many} = \{N.l.l, EMPTY.l.l\}\\&Relevant = \{EMPTY, EMPTY.l\}\end{aligned}$$

Cell $N.l.l$ is not suitable again, but also $EMPTY.l.l$ has the same values in both intlist and many. However, since the previous generation of cells had a difference (step 6), we add $EMPTY.l.l$ to the set of relevant cells:

$$Relevant = \{EMPTY, EMPTY.l, EMPTY.l.l\}$$

Using Algorithm 6.6.1, we can generate relevant cells for higher-order types. We shall never need to do this in its full generality, rather, for a higher-order type, we shall need to generate the initial set of relevant cells, and to find the relevant cells enabled by a state. Both these tasks are easy to accomplish using Definition 2.2.13 for the exponentiation dcds.

6.6.2 The algorithm

We are now ready to present our algorithm for refinement type inference for expressions.

Algorithm 6.6.3 (Refinement types for expressions) *Given an expression e, do the following to find its refinement type:*

1. *Find regular type τ for e.*

2. *If τ does not admit refinements, return it as the refinement type. Otherwise generate initial set I_τ of relevant cells of τ.*

3. *For each cell $c \in I_\tau$, ask the question e ? c.*

4. *If $e \mathrel{?} c \rightarrow v$, then add the event (c, v) to an approximation x of e. Find out what relevant cells are enabled by (c, v) and add them to I_τ.*

5. *If $e \mathrel{?} c \rightarrow valof\ c'$, then, if c' is a relevant cell, and it has a finite set of possible values, then for each possible value v', add $\{c' = v'\}c$ to I_τ.*

6. *If $e \mathrel{?} c \rightarrow loop$, then go on to the next cell in I_τ.*

7. *When the set I_τ is exhausted, apply Algorithm 6.3.1 to the approximation x.*

The algorithm always terminates because we are using the techniques of the previous section to evaluate $e \mathrel{?} c$, and because there are finitely many relevant cells.

The state x is clearly an approximation to the expression e. Since we apply Algorithm 6.3.1 to x to generate the refinement type, an algorithm whose soundness we have proven in Theorem 6.3.2, it follows that Algorithm 6.6.3 is also sound. Hence,

Theorem 6.6.4 *[Soundness of refinement type inference for expressions] If, according to Algorithm 6.6.3, $e :_r \tau$ then, indeed, $e :_r \tau$.*

Chapter 7

Implementation and Examples

In this chapter, we present an overview of our implementation and demonstrate the practical utility of our approach to refinement type inference with many examples. Section 7.1 covers the implementation, also pointing out differences with the theory of the previous two chapters. Section 7.2 contains CDS0 examples, and Section 7.3 PCF examples.

7.1 Implementation

Our prototype implementation of CDS0 with type inference and refinement type inference is in Standard ML of New Jersey, version 0.93. The implementation consists of about 15,000 lines of code, of which about 5,000 are automatically generated by YACC and LEX, or are part of the YACC base environment. The reason for the large number of lines of YACC and LEX code is that we have three interpreters as part of the system: a CDS0 interpreter, a PCF interpreter, and a cell name interpreter, for accepting user input in the request loop.

In the rest of this section, we shall attempt to give an idea of the structure of the CDS0 interpreter, and also describe our internal representation for certain notions presented in previous chapters.

7.1.1 Brief overview

Figure 7.1 shows a schematic diagram of the module dependencies in our implementation. Underlying the whole implementation are definitions of the CDS0 parse tree, CDS0 internal representations, CDS0 runtime environment, and PCF parse tree. The *Parser* module is a conglomeration of the three parsers and lexers already mentioned. *Printer* is a somewhat pretty printer, and *Match* performs the binding of cell and value variables during execution. *PcfCode* implements the translation to categorical combinators, and *Internal* the translation from CDS0 parse trees to forests. The translation to internal type representation (*idcds*) is in *Type*, which also implements algorithms for deciding subtyping for ground dcds's. Deciding subtyping for type expressions is done in *Subtype*, and the type inference of regular types is in *TypeChecker*. The *Evaluator* implements the CDS02 operational semantics. *QandA* is the questions and answers module, which generates relevant cells, and finds a relevant approximation to an expression. The *Refine* module puts together type inference and refinement type inference. Finally, *Toplevel* takes care of the top level loop, the request loop, and of error reporting.

Of some interest to the reader may be our implementation of forests, first described in Section 2.3.6. The datatype definitions are listed in Figure 7.2, omitting type definition of *tag*, which

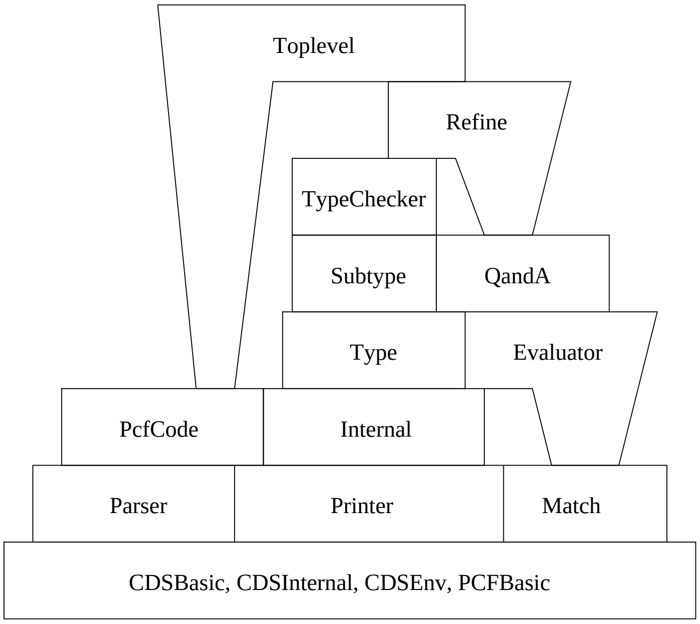

Figure 7.1: Module dependencies in our CDS0 implementation

is not relevant here. Typical of all our CDS0 definitions, the types of the internal representations of cells (*icell*), values (*ivalue*) and forests, are mutually recursive. This is due to the fact that we are implementing the CDS02 operational semantics, which allows expressions as part of a cell name. Hence, a cell name can be a name, a variable, a graft, a constrained name, or a *functional* name consisting of a forest and a cell name. The representation of tree instructions is just as in the theory. A basic forest is one given by enumeration of events, and is a pair of an integer and a list of trees; the integer specifies the degree of currification, *i.e.,* the number of inputs. For instance, a forest of type *int* will have degree 0, while a forest of type $int \rightarrow int \rightarrow int$ has degree 2.

We would also like to point out the internal representation used by the questions and answers module for the relevant portion of a type, which we call an *annotated* type:

```
datatype refineUnit = Ground of typeExp * typeExp list
                    | Ho of annotated
withtype outputRefinement = (int list * refineUnit) list
and inputRefinement = (int * int list * refineUnit) list
and annotated = int * outputRefinement * inputRefinement
```

A *typeExp* is a type expression, being either a ground dcds name, a variable, an arrow, and so on. The basic building block of an annotated type is a ground *refineUnit*, which lists the regular type name, and also a list of all refinements of that type. This is used to generate relevant cells. The integer lists in the intput and output refinement types keep track of the product tag, if any, of that piece of type. In addition, an *inputRefinement* also stores the index of the input it came from. Finally, an annotated type is a triple of an integer specifying the degree of currification, an *outputRefinement* and an *inputRefinement*.

```
datatype icell = Icell_name of string
               | Icell_var of string
               | Icell_fun of forest * icell
               | Icell_graft of icell * tag
               | Icell_with of icell * iboolexp
and ivalue = Ival_string of string
           | Ival_output of ivalue
           | Ival_valof of icell
           | Ival_arexpr of arexpr
           | Ival_omega
           | Ival_with of string * iboolexp
           | Ival_pair of ivalue * ivalue
and iboolexp = Iboolexp_gt of arexpr * arexpr
             | Iboolexp_gteq of arexpr * arexpr
             | Iboolexp_lt of arexpr * arexpr
             | Iboolexp_lteq of arexpr * arexpr
             | Iboolexp_eq of ivalue * ivalue
             | Iboolexp_noteq of ivalue * ivalue
             | Iboolexp_or of iboolexp * iboolexp
             | Iboolexp_and of iboolexp * iboolexp
and tree_instruction = tree_Valof of icell * int * tree_query list
                     | tree_From of icell * int * tree_query list
                     | tree_Result of int * ivalue
and forest = forest_basic of int * tree list
           | forest_apply of forest * forest
           | forest_comp of forest * forest
           | forest_fix of forest
           | forest_curry of forest
           | forest_uncurry of forest
           | forest_pair of forest list
           | forest_prod of forest list
withtype tree_query = ivalue * tree_instruction
and tree = icell * tree_instruction
```

Figure 7.2: Internal representation of forests

7.1.2 Differences between implementation and theory

The implementation of our CDS0 interpreter is a very close match to the theory previously developed. However, there are certain differences. None of them are fundamental in any theoretical sense; rather they are due to lack of time required for implementation. We enumerate the more important discrepancies below:

1. Overloaded types are treated like intersection types. Overloading is a side issue as far as refinement type inference is concerned. We have described how to handle it when presenting our type inference algorithm for CDS0 for the sake of completeness, because it is part of CDS0.

2. Not all dependencies between input and output variables (*cf.* Algorithm 6.3.1) are detected. In particular, those dependencies involving variables in arithmetic expressions are not detected. This would require a bit of machinery to implement fully, but it is not problematic from a theoretical point of view. It is not particularly relevant, because we have concentrated on examples in which this does not occur.

3. We treat certain types which have a mixture of polymorphic and non-polymorphic types as having a fully polymorphic type, even though they may still admit refinements. This could be detected, and the refinement type found, but its omission does not materially affect the kinds of examples we can handle.

7.2 CDS0 examples

Even though our implementation is a prototype, and no particular attention has been paid to fast execution, our results demonstrate that our approach to refinement type inference is practical. Unless specified, all examples presented below run in under one second. There are certain exceptions, however, which will be pointed out. When we mention running time, we are referring to our benchmark system, which is a Pentium Pro 200MHz with 256K L2 cache, 64 MB of RAM and 128 MB swap space, running Linux Red Hat 4.0. The running time is elapsed time.

Most of the CDS0 examples we present are algorithm definitions. This is due to the fact that the low-level nature of the language makes it difficult to write complex expressions. We begin with examples from Curien's book [26].

When the CDS0 interpreter starts up, it loads the base PCF environment, and leaves the user at the CDS0 prompt, denoted by #. This will be discussed in detail in the next section. Typing is optional so we turn it on, and define some of the types we have already encountered.

```
CDS0 version 1.1 --- June 11, 1997
# typing on;
# let bool = dcds cell B values tt,ff end;
Type bool defined.
# let int = dcds cell N values [..] end;
Type int defined.
# let true = dcds cell B values tt end;
Type true defined.
# let false = dcds cell B values ff end;
Type false defined.
# refine true, false;
```

Now very simple examples will have the expected types:

```
# {B=tt};
r: true
 : bool
request? ;
# {N=1};
r: int
 : int
request? ;
# {X=3};
Error: Type inference: term does not have a type
request? ;
```

Refinement typing judgments are denoted by "`r:`", and regular typing judgments by "`:`". The state $\{B = tt\}$ has the expected refinement type *true* and regular type *bool*, while $\{N = 1\}$ has *int* as both regular and refinement type. Note that typing in some state involving cells not occurring in any previously defined dcds results in a type error, but the interpreter permits evaluation to continue.

We arrive at more interesting results when we type in algorithms. We try this for boolean negation and left conjunction:

```
# let not =
  algo
    request B do
      valof B is
        tt : output ff
        ff : output tt
      end
    end
  end;
r: /\[false -> true, true -> false]
 : bool -> bool
Abbreviation "not" defined.
# let land =
  algo
    request B do
      valof (B.1) is
        tt: valof (B.2) is
              tt: output tt
              ff: output ff
            end
        ff: output ff
      end
    end
  end;
r: /\[(false * bool) -> false, (true * true) -> true, (true * false) -> false]
 : (bool * bool) -> bool
Abbreviation "land" defined.
```

Obtaining the refinement types above involves a simple application of Algorithm 6.3.1. The case of *land* involves instantiation of a type variable to *bool*, as discussed in the previous chapter.

We can define a curried version of *land* in two ways: one is the *curry_land* from Figure 2.5, but we can also apply *curry* to *land*, thus obtaining a combinator expression. Obtaining a refinement type for the combinator expression involves a different algorithm altogether than for *curry_land*, but, as expected, the types are the same:

```
# curry(land);
r: /\[false -> bool -> false, true -> false -> false, true -> true -> true]
 : bool -> bool -> bool
request? ;
# let curry_land =
  algo
    request {}B do
      valof B is
        tt: output valof B
        ff: output output ff
      end
    end
    request {B=tt}B do
      from {B=tt} do
        output output tt
      end
    end
    request {B=ff}B do
      from {B=tt} do
        output output ff
      end
    end
  end;
r: /\[false -> bool -> false, true -> false -> false, true -> true -> true]
 : bool -> bool -> bool
Abbreviation "curry_land" defined.
```

We can implement a right conjunction, and also a left strict conjunction. The interesting thing is that we can distinguish between these programs and left conjunction based on their refinement type:

```
# let rand = algo
  request B do
    valof (B.2) is
      tt: valof (B.1) is
            tt: output tt
            ff: output ff
          end
      ff: output ff
    end
  end
end;
```

```
r: /\[(bool * false) -> false, (true * true) -> true, (false * true) -> false]
 : (bool * bool) -> bool
Abbreviation "rand" defined.
# let lsand =
  algo
    request B do
      valof (B.1) is
        tt: valof (B.2) is
              tt: output tt
              ff: output ff
            end
        ff: valof (B.2) is
              tt: output ff
              ff: output ff
            end
      end
    end
  end;
r: /\[(false * false) -> false, (true * true) -> true, (false * true) -> false,
      (true * false) -> false]
 : (bool * bool) -> bool
Abbreviation "lsand" defined.
```

$rand$ has the refinement type $(bool \times false) \to false$, while $land$ has type $(false \times bool) \to false$. The idea is that we can use the refinement type to infer strictness information. $lsand$ does not have any refinement type involving $bool$.

One kind of program that we can write in CDS0 but not in PCF is an algorithm that does semantic manipulation. By this we mean that the algorithm does different things depending on how its input reacts to various inputs. A fascinating example of such an algorithm is called AND_TASTER, and was first described by Berry and Curien [7]. The algorithm takes as input an algorithm on $(bool \times bool) \to bool$ and determines if it is a conjunction algorithm, and if so which one. The full text of the algorithm is given in Appendix B.5. Here we define the type of its output, together with refinements.

```
# let and_type =
  dcds
    cell WHICH_AND values IS_LEFT_AND, IS_LEFT_STRICT_AND,
                          IS_RIGHT_AND, IS_RIGHT_STRICT_AND,
                          IS_NOT_AN_AND
  end;
Type and_type defined.
# let is_and_type =
  dcds
    cell WHICH_AND values IS_LEFT_AND, IS_LEFT_STRICT_AND,
                          IS_RIGHT_AND, IS_RIGHT_STRICT_AND
  end;
Type is_and_type defined.
```

```
# let is_not_and_type =
  dcds
    cell WHICH_AND values IS_NOT_AN_AND
  end;
Type is_not_and_type defined.
# refine is_and_type, is_not_and_type;
```

Given these refinements, AND_TASTER has an incredibly detailed type, which is also listed in Appendix B.5. In this case, the intensional information is overwhelming: there is a type for each possible branch through the program, and so the refinement type ends up being not much more succinct than the code itself.

We now turn our attention to the implementation of lazy natural numbers we presented earlier, and show how our refinement types helped catch an error. We begin by defining lazy natural numbers and two refinements, empty, and non-empty:

```
# letrec lnat =
  dcds
    cell B values 0,1
    graft (lnat.s) access B = 1
  end;
Type lnat defined.
# let empty_lnat = dcds
  cell B values 0
end;
Type empty_lnat defined.
# local letrec partial_lnat = dcds
        cell (B.s) values 0,1 access B = 1
        graft (partial_lnat.s) access B = 1
      end
  in let some_lnat = dcds
      cell B values 1
      cell (B.s) values 0,1 access B = 1
      graft (partial_lnat.s)
    end
end;
Type some_lnat defined.
# refine empty_lnat, some_lnat;
```

Some of the algorithms on lazy natural numbers presented earlied have the expected refinement types. The refinement type of successor can be simplified by removing the middle type, but our interpreter currently does not handle this.

```
# let Somega = fix(Srec);
r: some_lnat
 : lnat
Abbreviation "Somega" defined.
# let S = fix(succ_rec);
r: /\[some_lnat -> some_lnat, lnat -> some_lnat, empty_lnat -> some_lnat]
 : lnat -> lnat
Abbreviation "S" defined.
```

It turns out that our first implementation of left minimum was erroneous, in that if it received a 0 in the left input, it still checked the right input, instead of placing a 0 on the output right away. Given a right input which looped, this program, of course, would loop in that situation. We did not catch this error until we implemented refinement type inference, and observed the following type for the program:

```
# let bad_left_min = fix(bad_left_min_rec);
r: /\[(some_lnat * some_lnat) -> some_lnat,
      (empty_lnat * empty_lnat) -> empty_lnat,
      (some_lnat * empty_lnat) -> empty_lnat,
      (empty_lnat * some_lnat) -> empty_lnat]
 : (lnat * lnat) -> lnat
Abbreviation "bad_left_min" defined.
```

This provided a clue that the program did not do what we intended. The revised, correct program, shown in Appendix B.1, has the expected type:

```
# let left_min = fix(left_min_rec);
r: /\[(some_lnat * some_lnat) -> some_lnat, (empty_lnat * lnat) -> empty_lnat,
      (some_lnat * empty_lnat) -> empty_lnat]
 : (lnat * lnat) -> lnat
Abbreviation "left_min" defined.
```

7.3 PCF

We mentioned previously that upon startup the CDS0 interpreter loads in the base PCF environment. This consists of the type definitions and the combinators required to compile PCF to CDS0. There are actually two base environments for PCF that we implemented: one contains refinement types and one does not. We first discuss the types obtained for the terms in the base environments before turning our attention to PCF.

7.3.1 Base environment

There are two ways of starting up the CDS0 interpreter: regular or refinement typing. The choice of typing only applies to PCF. We discuss refinement typing, since it subsumes regular typing. The complete listing of CDS0 programs which make up the refinement compilation environment is given in Appendix B.3. The complete transcript of the interpreter processing the CDS0 programs is given in Appendix B.4. Here we discuss an abbreviated list. Note that the refinement type and regular type are listed before the name of the algorithm.

```
- cds0(refined);
-- Loading PCF constants.
Type bool defined.
Type int defined.
Type true defined.
Type false defined.
r: /\[((false * 'a) * 'b) -> 'b, ((true * 'c) * 'd) -> 'c]
 : ((bool * 'a) * 'a) -> 'a
Abbreviation "cond" defined.
```

```
r: ('a * 'b) -> 'a
 : ('a * 'b) -> 'a
Abbreviation "fst" defined.
r: ('a * 'b) -> 'b
 : ('a * 'b) -> 'b
Abbreviation "snd" defined.
r: (int * int) -> int
 : (int * int) -> int
Abbreviation "plus" defined.
...
r: (int * int) -> bool
 : (int * int) -> bool
Abbreviation "equal" defined.
...
r: 'a -> 'a
 : 'a -> 'a
Abbreviation "id" defined.
r: 'a
 : 'a
Abbreviation "emptyenv" defined.
r: ('a -> 'a) -> 'a
 : ('a -> 'a) -> 'a
Abbreviation "Y" defined.
r: ('a -> 'b -> 'b) -> 'a -> 'b
 : ('a -> 'b -> 'b) -> 'a -> 'b
Abbreviation "Yenv" defined.
Type intlist defined.
Type empty_intlist defined.
Type one_intlist defined.
Type many_intlist defined.
r: empty_intlist
 : intlist
Abbreviation "nil" defined.
r: /\[one_intlist -> false, empty_intlist -> true, many_intlist -> false]
 : intlist -> bool
Abbreviation "null" defined.
r: /\[(int * one_intlist) -> many_intlist, (int * many_intlist) -> many_intlist,
      (int * empty_intlist) -> one_intlist]
 : (int * intlist) -> intlist
Abbreviation "cons" defined.
r: /\[one_intlist -> int, many_intlist -> int]
 : intlist -> int
Abbreviation "hd" defined.
r: /\[many_intlist -> intlist, one_intlist -> empty_intlist]
 : intlist -> intlist
Abbreviation "tl" defined.
```

The first interesting type to observe is that for conditional. When collecting cell names and values along the two possible paths to an output through *cond*, we can establish a dependence between the variable cell names and values: when input is $true$, the output cell $\$C$, gets its value from the left input and when the input is $false$, it gets its value from the right input. By unifying the matching type variables, we can obtain the more precise refinement type.

The identity, projections, Y and Y_{env} combinators have fully polymorphic types, and so do not admit refinements. Also having a polymorphic type is the empty environment, which is implemented as the empty state. The reason for this is that the empty state can be part of any dcds.

We cannot obtain interesting refinement types for the addition and equality check for integers algorithms presented in the previous chapter. The type of addition does not admit refinements. As for the equality test, the types of its two paths are $(int \times int) \rightarrow true$ and $(int \times int) \rightarrow false$. In the last step of Algorithm 6.3.1, we make this into $(int \times int) \rightarrow bool$, because we must lose precision.

Finally, the algorithms on integer lists have the expected types, as discussed in the previous chapter.

7.3.2 Examples

We can switch from the CDS0 interpreter to the PCF one by typing the command `pcf`. At this time, the prompt changes to `$` and the base enviroment for PCF becomes the current environment. The refinement types for all of the examples in this section are obtained through the use of Algorithm 6.6.3, by entering a questions and answers session with the expression which exposes a relevant state, which is then typed using Algorithm 6.3.1.

We begin with two examples on $bool \rightarrow bool$: boolean negation and a function which always returns true.

```
$ val not = fn x => if x then false else true;
r: /\[false -> true, true -> false]
 : bool -> bool
Abbreviation "not" defined.
$ val exclmid = fn x => x or (not x);
r: /\[false -> true, true -> true]
 : bool -> bool
Abbreviation "exclmid" defined.
```

The interesting thing to note here is that this version of *not* is implemented in a completely different fashion than the version we wrote in CDS0 directly (*cf.* Section 6.4.2). As expected, we obtain the same refinement type.

Now we consider some programs on integer lists. The *map* function we first presented in the introduction has the refinement type we had wanted:

```
val map = letrec mapf = fn f => fn l =>
                        if null l then [] else (f (hd l)) :: ((mapf f) (tl l))
          in mapf
          end;
r: /\[(int -> int) -> many_intlist -> many_intlist,
      (int -> int) -> one_intlist -> one_intlist,
      (int -> int) -> empty_intlist -> empty_intlist]
 : (int -> int) -> intlist -> intlist
Abbreviation "map" defined.
```

One of the strengths of our approach to refinement type inference over the Freeman-Pfenning one can be seen in the following examples:

```
$ val l3 = 1 :: (2 :: (3 :: []));
r: many_intlist
 : intlist
Abbreviation "l3" defined.
$ tl l3;
r: many_intlist
 : intlist
request? ;
$ tl (tl l3);
r: one_intlist
 : intlist
```

Because we are not using type inference rules, we are able to obtain more precise refinement types. Recall that one of the refinement types of *tl* is *many_intlist* $\rightarrow$ *intlist*, a type which entails inevitable loss of precision. By using such a type with inference rules it is impossible to obtain anything other than *intlist* for the refinement types of the two expressions above. Since we query the expression directly, we bypass this problem.

Another advantage of our approach is that we place no restrictions on polymorphic functions. When presenting the Freeman-Pfenning approach in Section 2.6, we described how, in the case of an example such as *double not*, the function *double* could not be polymorphic. Even then, obtaining the refinement type was complicated by instantiations of type variables which led to very long refinement types. In our case, the answer can be obtained very quickly; the questions and answers session only needs to know the values of three cells in order to give a precise refinement type:

```
$ val double = fn f => fn x => f (f x);
r: ('a -> 'a) -> 'a -> 'a
 : ('a -> 'a) -> 'a -> 'a
Abbreviation "double" defined.
$ double not;
r: /\[false -> false, true -> true]
 : bool -> bool
```

When performing the questions and answers session in Algorithm 6.6.3, we cannot construct new queries when we need to know the values of cells that have an infinite number of possible values. Depending on the flow of control we may or may not be able to obtain precise types for expressions with inputs that have such cells:

```
$ val f = fn x => fn y => fn l => (x+y)::l;
r: /\[int -> int -> many_intlist -> many_intlist,
      int -> int -> one_intlist -> many_intlist,
      int -> int -> empty_intlist -> one_intlist]
 : int -> int -> intlist -> intlist
Abbreviation "f" defined.
$ val h = fn x => fn l => if x=3 then l else x::l;
r: int -> intlist -> intlist
```

```
 : int -> intlist -> intlist
Abbreviation "h" defined.
```

In the first example above, we obtain something very precise, and very similar to the type of *cons*. In the second example we are stuck, since we need to know the value of the input, which comes from *int*. We cannot construct a relevant approximation, so we return the regular type as the refinement type.

Even when faced with rather complicated-looking expressions, we can usually obtain a refinement type very quickly. Our system infers a precise type for $test_2$ below with no perceptible wait time. *useless* is a function which always returns an empty list.

```
$ val useless = letrec f = fn l => if null l then [] else f (tl l)
              in f
              end;
r: /\[many_intlist -> empty_intlist, empty_intlist -> empty_intlist,
      one_intlist -> empty_intlist]
 : intlist -> intlist
Abbreviation "useless" defined.
$ val times2 = fn x => x * 2;
r: int -> int
 : int -> int
Abbreviation "times2" defined.
$ val test2 = 1 :: (useless ((map times2) (useless ((map times2) l3))));
r: one_intlist
 : intlist
Abbreviation "test2" defined.
```

Sometimes we are only able to obtain partial information. Consider the following example of the familiar predicate *exists*:

```
$ val exists = letrec f = fn p => fn l =>
                 if null l then false
                 else if p (hd l) then true
                                  else (f p) (tl l)
                in f end;
r: /\[(int -> false) -> empty_intlist -> false,
      (int -> true) -> empty_intlist -> false,
      (int -> bool) -> empty_intlist -> false]
 : (int -> bool) -> intlist -> bool
Abbreviation "exists" defined.
```

Again, the type could have been simplified by removing the last branch. We are only able to infer what happens if the input is empty. When querying the expression with a non-empty input, we receive a $valof\ xc$ answer, where x is a complex expression asking, in essence, if the property holds of the input. We cannot do anything with such an answer, so we give up. This brings us to one of the weaknesses of our approach: we are not always able to obtain precise refinement types for expressions with higher-order inputs. Refinement type inference rules, as in the Freeman-Pfenning approach, would work better in such cases. As an example, consider the following program:

```
$ val f11 = letrec f = fn b => fn g =>
              if g b then true else (f false) g in f end;
r: bool -> (bool -> bool) -> bool
 : bool -> (bool -> bool) -> bool
Abbreviation "f11" defined.
```

By making assumptions about all possible refinement typings of g, a system with refinement type inference rules might be able to obtain more precise types.

We can obtain very detailed types very quickly, for certain many recursive functions, such as *append*.

```
$ val append = letrec f = fn l1 => fn l2 => if null l1 then l2
                                           else (hd l1) :: ((f (tl l1)) l2)
             in f end;
r: /\[many_intlist -> intlist -> many_intlist,
     many_intlist -> one_intlist -> many_intlist,
     many_intlist -> many_intlist -> many_intlist,
     many_intlist -> empty_intlist -> many_intlist,
     one_intlist -> many_intlist -> many_intlist,
     one_intlist -> one_intlist -> many_intlist,
     one_intlist -> empty_intlist -> one_intlist,
     empty_intlist -> many_intlist -> many_intlist,
     empty_intlist -> one_intlist -> one_intlist,
     empty_intlist -> empty_intlist -> empty_intlist]
 : intlist -> intlist -> intlist
Abbreviation "append" defined.
```

However, we have encountered recursive functions where the performance of refinement type inference suffers from the limitations of CDS02.

```
$ val rev1 = letrec f = fn l => fn result =>
               if null l then result
               else (f (tl l)) ((hd l) :: result)
             in f end;
...
 : intlist -> intlist -> intlist
Abbreviation "rev1" defined.
$ val rev = fn l => (rev1 l) [];
r: /\[many_intlist -> many_intlist, empty_intlist -> empty_intlist,
      one_intlist -> one_intlist]
 : intlist -> intlist
Abbreviation "rev" defined.
```

We have omitted the refinement type for rev_1 because it is very similar to that of *append*. Obtaining the refinement type for rev_1 takes 6 seconds, and for rev 16 seconds. These are the only programs presented so far on which our interpreter takes more than a second. The poor performance is due to the fact that the computation of fixpoints is not memoized in CDS02.

We end with some looping programs, one of which has a type which does not admit refinements, hence it is "sidestepped," one that is detected, and one that reaches the depth bound.

```
$ val loop2 = letrec f = fn x => f (x+1) in f end;
r: int -> 'a
 : int -> 'a
Abbreviation "loop2" defined.
$ val loop6 = letrec f = fn l => if null l then [] else f l in f end;
This expression loops.
r: intlist -> intlist
 : intlist -> intlist
Abbreviation "loop6" defined.
$ val loop8 = letrec f = fn l => if null l then [] else f (1::l) in f end;
This expression may loop.  Refinement type inference gives up.
r: intlist -> intlist
 : intlist -> intlist
Abbreviation "loop8" defined.
```

Chapter 8

Conclusions and Further Work

In this chapter we conclude and present possible avenues for further work.

8.1 General conclusions

We believe we have provided ample evidence of both the theoretical and practical utility of studying intensional semantics. Thus, the central claim of the thesis has been demonstrated. However, the central claim was very broad, so we present a more detailed assessment of this work.

8.1.1 Relative intensional expressiveness

We defined the notion of relative intensional expressiveness between programming languages, developed a new intensional semantics, circuit semantics, and we set out to prove separation results. Our goal was to compare languages, and not underlying computation models. We have been able to compare primitive recursive algorithms with sequential algorithms and parallel algorithms, PCF extended with por, pif_o, pif_ι and deterministic $query$, and, finally, PCF extended with deterministic and non-deterministic query. However, in the process, we have been only partially successful in staying true to our original goal of only comparing languages. Of the comparisons we have made, three rely to some extent on assumptions about computation models:

1. When comparing CDS0 and CDSP, we allowed a construct to evaluate cells in parallel in CDSP, but CDS0, due to the inherently sequential nature of its operational semantics, could not do something similar. Thus, our comparison became partly a comparison of a sequential and a parallel machine model.

2. Similarly, when comparing PCF extended with pif_ι versus query, we only allowed parallel computations to be started by query, or the limited mechanism of pif_ι. This made the comparison somewhat artificial, because there are possible parallel evaluation styles for PCF as a whole.

3. The most egregious break with our original goal was made when comparing deterministic and non-deterministic query. In order to obtain deterministic results, we focused on a subset of non-deterministic queries which return deterministic answers under the assumption of hardware that detects undefined inputs. Deterministic queries cannot take advantage of this hardware. Using results from circuit complexity, we were then able to prove that non-deterministic query is more expressive. But we believe the insight gained into connections

between complexity theory and programming languages theory, and in particular, between DPCF and monotone circuits, offsets the shortcomings of the method. Note that this connection still applies if we add recursion to DPCF.

The major lesson we have learned from this is that it is interesting and worthwhile to attempt intensional comparisons of programming languages, but that it is difficult to achieve it in a completely fair fashion.

8.1.2 Refinement type inference

We developed a novel type inference system based on concrete data structures, which have a more elaborate structure than records, and we implemented it in our CDS0 interpreter. Gradually, we realized that we could type the various paths through an algorithm separately and achieve what was called in the literature a refinement type. After becoming aware of the work of Hughes and Ferguson, we developed our questions and answers approach to refinement type inference, which has benefits and drawbacks as compared to the only previous approach, that of Freeman and Pfenning. The benefits are:

1. No restrictions are placed on the usage of polymorphic functions in order to obtain precise refinement types.
2. No need to consider all possible refinements of a type, which leads to large time and space savings, especially in the case of higher-order types.
3. Ability to obtain more precise refinement types in many cases.

The drawbacks of our approach are:

1. Poor performance in the case of certain kinds of fixpoint computations, due to the underlying CDS02 semantics.
2. Inability to obtain precise refinement types in many cases, especially when higher-order inputs are involved.
3. A more restrictive language for defining refinement types. In particular, we cannot have definitions such as the even and odd refinements of boolean lists of Section 2.6.

There are other differences between the two systems, but they are not as important, For instance, we do not have polymorphic lists. This can be easily remedied, however. As we have seen, the relevant cells in a list, from the point of view of refinement type inference, are the backbone cells, $EMPTY$, $EMPTY.l$, and $EMPTY.l.l$. Regardless of which kinds of lists we considered, those cells would remain the same, thus we can imagine extending CDS0 with "generic" list definitions.

Despite the drawbacks, we believe our system shows signs of being quite practical. We have already benefited from it in finding a programming error. There are two obstacles that we see before the system becomes truly practical: the type definitions must currently be done in CDS0, and there exist performance concerns for certain recursive functions. We believe these problems can be solved, and we shall discuss this issue in the next section.

In the process of developing the refinement type inference system, we established a new way of using sequential algorithms to perform abstract interpretation. The previous approach, that of Hughes and Ferguson, suffered from severe space problems [51]. Our approach has speed problems in the case of fixpoints. The natural question is how to combine the best of the two approaches. This possibility is also discussed in the next section.

8.2 Further work

We discuss possible areas of further work in three main categories: refinement type inference, CDS0 applications, and extensions of CDS0.

8.2.1 Refinement type inference

The obvious idea suggested by the comments in the previous section, is to have a mixed refinement type inference rules and abstract interpretation of the expression approach. Consider the following program:

```
rev (tl ones);
```

where *ones* is the infinite list of 1's defined earlier. Our system can infer a precise type for a piece of the program:

```
$ tl ones;
r: many_intlist
 : intlist
request? ;
```

The Freeman-Pfenning system can only infer *intlist* as the type of this expression. However, given something known to have refinement type *many_intlist*, that system could obtain type *many_intlist* for the result of applying *rev* to it. Our system cannot infer *many_intlist* as the final answer, because the computation loops, so refinement type inference gives up. A combined system would be able to obtain the type *many_intlist* for the whole expression. Of course, it is not clear how to achieve this combination of the two approaches in detail, but it seems like a particularly interesting area for future work.

There are several ways in which our implementation can be improved. Aside from removing the discrepancies between the implementation and the theory, there are two main ways we could strive for better performance:

1. Simplification of categorical code. Currently, we perform no simplifications at all on the combinator code which results from the compilation of PCF programs. This code is very inefficient. One of the major implementations of ML, CAML [61, 25], is based on the same compilation to categorical combinators, and it relies on many optimizations. Adopting even a small subset of these for our purposes would probably result in markedly improved performance.

2. Memoization of fixpoint computations in CDS02. We believe this is the main performance bottleneck. Developing a mixed CDS02/01 evaluation strategy that keeps tables around for fixpoints should solve most of our performance problems.

As far as having to define types in CDS0 is concerned, we believe this is not an enormous problem. Having to write programs in CDS0 is more of a concern, but, as we have shown, that can be avoided. It is possible to make the CDS0 type definitions more like to ML-style definitions. The original paper on CDS0 [5] takes some steps in this direction, and one can probably go much further.

8.2.2 Applications of CDS0

We have already mentioned that we believe CDS02 confers significant advantages for the purpose of abstract interpretation over CDS01. It seems Hughes and Ferguson have considered the space problems of CDS01 insurmountable [51]. We plan on developing strictness analysis based on CDS02, and hope to see the same great performance mentioned in [33] without the massive storage use.

Another area we plan to investigate is the use of CDS02 profiling semantics for the purpose of complexity analysis of lazy, higher-order programs. We have already made a start in Chapter 3, by providing operational semantics rules extended with step information. It would be quite interesting to analyze some of the problems in Wadler [85] and Sands [78] with our approach. One of the great strengths of sequential algorithms is that they provide a uniform way of moving from first-order to higher-order, and so a lot of the problems encountered in the previously cited approaches might disappear.

Finally, CDS0 is an implementation of a game semantics, and much has been written about connections between game semantics and parallel implementations of functional languages (see [1], for instance). The idea is that a sequential computation can be broken down into a network of concurrent processes which exchange information. We are of the opinion that our refinement type inference framework can be extended for the purpose of analyzing such networks of processes communicating through channels, and we have already started work in this direction.

8.2.3 Extensions of CDS0

We briefly mention two ideas on extensions of CDS0, which are somewhat distantly related to our current work. First, we are considering the possibility of allowing non-ground dcds definitions, and extending the language with channels. The idea would be to be able to send higher-order messages along a channel in a piece-meal fashion. It is not clear yet how this extension would affect the type system we developed for CDS0.

Second, we are envisioning a parallel extension of CDS0 for artificial intelligence applications. One of the most interesting features of CDS0 is the ability to write semantics-manipulation algorithms, such as AND_TASTER. Imagine a language for programming virtual worlds in which agents can meet and interact based on each other's semantics. We could use refinement types in such a system to obtain very interesting behavioral information on agents.

Appendix A

Summary of Major Definitions

A.1 CDS0 operational semantics

A.1.1 Evaluation of forests

(TREE1) $$\frac{c'_i = c'}{\mathit{Tree}\ (c'_1, \mathit{ins}_1), \ldots, \mathit{Tree}\ (c'_n, \mathit{ins}_n)\ ?\ x_1 \cdots x_n c' \ \rightarrow\ \mathit{ins}_i\ ?\ x_1 \cdots x_n c'}$$

(TREE2) $$\frac{\forall i.\ c'_i \neq c'}{\mathit{Tree}\ (c'_1, \mathit{ins}_1), \ldots, \mathit{Tree}\ (c'_n, \mathit{ins}_n)\ ?\ x_1 \cdots x_n c' \ \rightarrow\ \Omega}$$

(RESULT) $$\mathit{Result}\ v'\ ?\ x_1 \cdots x_n c' \ \rightarrow\ v'$$

(VALOF) $$\frac{x_p\ ?\ c\ \rightarrow \left\{\begin{array}{l} v_i \\ v,\ \text{and}\ \forall i.\ v_i \neq v \\ \Omega \end{array}\right.}{\left.\begin{array}{l} \mathit{Valof}\ \ (c,p)\ \ \mathit{is} \\ \ \ v_1\ :\ \mathit{ins}_1 \\ \ \ \ldots \\ \mathit{end} \end{array}\right\}\ ?\ x_1 \cdots x_n c'\ \rightarrow \left\{\begin{array}{l} \mathit{ins}_i\ ?\ x_1 \cdots x_n c' \\ \Omega \\ \mathit{output}^{p-1}\ \mathit{valof}\ c \end{array}\right.}$$

(FROM) $$\frac{x_p\ ?\ c\ \rightarrow \left\{\begin{array}{l} v_i \\ v,\ \text{and}\ \forall i.\ v_i \neq v \\ \Omega \end{array}\right.}{\left.\begin{array}{l} \mathit{From}\ \ (c,p)\ \ \mathit{is} \\ \ \ v_1\ :\ \mathit{ins}_1 \\ \ \ \ldots \\ \mathit{end} \end{array}\right\}\ ?\ x_1 \cdots x_n c'\ \rightarrow \left\{\begin{array}{l} \mathit{ins}_i\ ?\ x_1 \cdots x_n c' \\ \text{fail with no-access} \\ \Omega \end{array}\right.}$$

A.1.2 CDS02 rules

(APP) $$\frac{A\ ?\ Bc' \rightarrow \left\{\begin{array}{c} \Omega \\ \mathit{valof}\ c \\ \mathit{output}\ v' \end{array}\right.}{A.B\ ?\ c' \rightarrow \left\{\begin{array}{c} \Omega \\ \Omega \\ v' \end{array}\right.}$$

(Comp)
$$\frac{A\ ?\ (B.x)c'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ c' & B?xc' \rightarrow \mathit{valof}\ c \\ \mathit{output}\ v'' \end{cases}}{A|B\ ?\ xc'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ c \\ \mathit{output}\ v' \end{cases}}$$

(Fix)
$$\frac{A\ ?\ fix(A)c \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ c' \\ \mathit{output}\ v \end{cases}}{fix(A)\ ?\ c \rightarrow \begin{cases} \Omega \\ \Omega \\ v \end{cases}}$$

(Pair)
$$< A_1, \ldots, A_n >\ ?\ x(c.i) \rightarrow A_i\ ?\ xc$$

(Prod)
$$\prod_{i=1}^{n} A_i\ ?\ (c.i) \rightarrow A_i\ ?\ c$$

(Curry)
$$\frac{A\ ?\ (x \times y)c'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ (c.1) \\ \mathit{valof}\ (c'.2) \\ \mathit{output}\ v'' \end{cases}}{curry(A)\ ?\ xyc'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ c \\ \mathit{output}\ \mathit{valof}\ c' \\ \mathit{output}\ \mathit{output}\ v'' \end{cases}}$$

(Uncurry)
$$\frac{A\ ?\ (\pi_1.x)(\pi_2.y)c'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ c \\ \mathit{output}\ \mathit{valof}\ c' \\ \mathit{output}\ \mathit{output}\ v'' \end{cases}}{uncurry(A)\ ?\ xc'' \rightarrow \begin{cases} \Omega \\ \mathit{valof}\ (c.1) \\ \mathit{valof}\ (c'.2) \\ \mathit{output}\ v'' \end{cases}}$$

A.2 CDS0 typing rules

A.2.1 Subtyping and intersection types

(Sub-Refl)
$$\sigma \leq \sigma$$

(Sub-Trans)
$$\frac{\sigma \leq \tau \quad \tau \leq \delta}{\sigma \leq \delta}$$

(Sub-Prod)
$$\frac{\sigma_1 \leq \tau_1 \quad \sigma_2 \leq \tau_2}{\sigma_1 \times \sigma_2 \leq \tau_1 \times \tau_2}$$

(AND-INTRO) $$\frac{x : \sigma_1 \ \cdots \ x : \sigma_n}{x : \bigwedge[\sigma_1..\sigma_n]}$$

(AND-ELIM) $$\frac{x : \bigwedge[\sigma_1..\sigma_n]}{x : \sigma_i}$$

(SUB-AND-R) $$\frac{\forall i.\ \sigma \leq \tau_i}{\sigma \leq \bigwedge[\tau_1..\tau_n]}$$

(SUB-AND-L) $$\bigwedge[\sigma_1..\sigma_n] \leq \sigma_i$$

(SUB-ARROW) $$\frac{\sigma_2 \leq \sigma_1 \qquad \tau_1 \leq \tau_2}{\sigma_1 \rightarrow \tau_1 \leq \sigma_2 \rightarrow \tau_2}$$

(SUB-AND-DIST) $$\bigwedge[\sigma \rightarrow \tau_1 \ .. \ \sigma \rightarrow \tau_n] \leq \sigma \rightarrow \bigwedge[\tau_1..\tau_n]$$

(SUB-OVER) $$\{\sigma_i \rightarrow \tau_i \mid i \in 1..n\} \leq \{\delta_j \rightarrow \zeta_j \mid j \in 1..m\} \qquad \forall j.\ \exists i.\ \sigma_i \rightarrow \tau_i \leq \delta_j \rightarrow \zeta_j$$

(SUB-MEET-OVER) $$\bigwedge[\sigma_i \rightarrow \tau_i \mid i \in 1..n] \leq \{\sigma_i \rightarrow \tau_i \mid i \in 1..n\}$$

A.2.2 Monomorphic type inference

(APP) $$\frac{a : \bigwedge[\sigma_i \rightarrow \tau_i \mid i \in 1..n] \quad b : \bigwedge[\delta_1..\delta_m]}{a.b : \bigwedge[\tau_i \mid \exists j.\ \delta_j \leq \sigma_i]}$$

(COMP) $$\frac{a : \bigwedge[\tau_i \rightarrow \delta_i \mid i \in 1..n] \quad b : \bigwedge[\sigma_j \rightarrow \tau'_j \mid j \in 1..m]}{a|b : \bigwedge[\sigma_j \rightarrow \delta_i \mid \tau'_j \leq \tau_i]}$$

(FIX) $$\frac{a : \bigwedge[\sigma_i \rightarrow \tau_i \mid i \in 1..n]}{fix(a) : \bigwedge[\sigma_i \mid \sigma_i \geq \tau_i]}$$

(CURRY) $$\frac{a : \bigwedge[(\sigma_i \times \sigma'_i) \rightarrow \tau_i \mid i \in 1..n]}{curry(a) : \bigwedge[\sigma_i \rightarrow \sigma'_i \rightarrow \tau_i \mid i \in 1..n]}$$

(UNCURRY) $$\frac{a : \bigwedge[\sigma_i \rightarrow \sigma'_i \rightarrow \tau_i \mid i \in 1..n]}{uncurry(a) : \bigwedge[(\sigma_i \times \sigma'_i) \rightarrow \tau_i \mid i \in 1..n]}$$

(PAIR) $$\frac{a : \bigwedge[\sigma_i \rightarrow \tau_i \mid i \in 1..n] \quad b : \bigwedge[\delta_j \rightarrow \zeta_j \mid j \in 1..m]}{< a, b > : \bigwedge[\sigma_i \rightarrow (\tau_i \times \zeta_j) \mid \delta_j \leq \sigma_i]}$$

(PROD) $$\frac{a : \bigwedge[\tau_1..\tau_n] \quad b : \bigwedge[\tau'_1..\tau'_m]}{(a, b) : \bigwedge[\tau_i \times \tau'_j \mid i \in 1..n,\ j \in 1..m]}$$

A.2.3 Polymorphic type inference

(GEN) $$\frac{e : \sigma}{e : \forall\alpha.\ \sigma}$$

(INST) $$\frac{e : \forall\alpha.\ \sigma}{e : [\tau/\alpha]\sigma}$$

(APP-OVER) $$\frac{a : \{\sigma_i \rightarrow \tau_i \mid i \in 1..n\} \quad b : \sigma}{a.b : \bigvee[\tau_i \mid \sigma \leq \sigma_i]}$$

(Comp-Over) $$\frac{a : \{\tau_i \to \delta_i \mid i \in 1..n\} \quad b : \sigma \to \tau'}{a|b : \sigma \to \bigvee[\delta_i \mid \tau' \leq \tau_i]}$$

(Fix-Over) $$\frac{a : \{\sigma_i \to \tau_i \mid i \in 1..n\}}{fix(a) : \bigvee[\sigma_i \mid \sigma_i \geq \tau_i]}$$

(Curry-Over) $$\frac{a : \{(\sigma_i \times \sigma_i') \to \tau_i \mid i \in 1..n\}}{curry(a) : \{\sigma_i \to \sigma_i' \to \tau_i \mid i \in 1..n\}}$$

(Uncurry-Over) $$\frac{a : \{\sigma_i \to \sigma_i' \to \tau_i \mid i \in 1..n\}}{uncurry(a) : \{(\sigma_i \times \sigma_i') \to \tau_i \mid i \in 1..n\}}$$

A.2.4 Refinement types

(Ref-Refl) $$\tau \sqsubseteq \tau$$

(Ref-Sub) $$\frac{\sigma \leq_p^* \tau}{\sigma \sqsubseteq \tau}$$

(Ref-And) $$\bigwedge[\sigma_1..\sigma_n] \sqsubseteq \bigwedge[\tau_1..\tau_m] \qquad \forall i\ \exists j.\ \sigma_i \sqsubseteq \tau_j$$

(Ref-Arrow) $$\frac{\sigma_1 \sqsubseteq \sigma_2 \quad \tau_1 \sqsubseteq \tau_2}{\sigma_1 \to \tau_1 \sqsubseteq \sigma_2 \to \tau_2}$$

(Ref-Prod) $$\frac{\sigma_1 \sqsubseteq \sigma_2 \quad \tau_1 \sqsubseteq \tau_2}{\sigma_1 \times \tau_1 \sqsubseteq \sigma_2 \times \tau_2}$$

Appendix B

CDS0 and CDSP Algorithms

B.1 left_min

```
let left_min_rec = algo
  request {}B do
    output valof (B.1)
  end
  request {(B.1)=0}B do
    output output 0
  end
  request {(B.1)=1}B do
    output valof (B.2)
  end
  request {(B.1)=1,(B.2)=0}B do
    output output 0
  end
  request {(B.1)=1,(B.2)=1}B do
    output output 1
  end
  request {}((B.$V).s) do
    valof {}(B.$V) is
      valof ((B.$V).1) : output valof (((B.$V).s).1)
    end
  end
  request {(((B.$V).s).1)=0}((B.$V).s) do
    from {{}(B.$V)=valof ((B.$V).1)} do
      valof {((B.$V).1)=0}(B.$V) is
        output 0 : output output 0
      end
    end
  end
  request {(((B.$V).s).1)=1}((B.$V).s) do
    from {{}(B.$V)=valof ((B.$V).1)} do
      valof {((B.$V).1)=1}(B.$V) is
        valof ((B.$V).2) : output valof (((B.$V).s).2)
      end
    end
  end
  request {(((B.$V).s).1)=1,(((B.$V).s).2)=0}((B.$V).s) do
```

```
    from {{}(B.$V)=valof ((B.$V).1),{((B.$V).1)=1}(B.$V)=valof ((B.$V).2)} do
      valof {((B.$V).1)=1,((B.$V).2)=0}(B.$V) is
        output 0 : output output 0
      end
    end
  end
  request {(((B.$V).s).1)=1,(((B.$V).s).2)=1}((B.$V).s) do
    from {{}(B.$V)=valof ((B.$V).1),{((B.$V).1)=1}(B.$V)=valof ((B.$V).2)} do
      valof {((B.$V).1)=1,((B.$V).2)=1}(B.$V) is
        output 1 : output output 1
      end
    end
  end
end;

let left_min = fix(left_min_rec);
```

B.2 min

```
let min_rec = algo
  request {}B do
    output query {(B.1), (B.2)} is
             {0, _} => output 0
             {_, 0} => output 0
             {1, 1} => output 1
           end
  end
  request {}((B.$V).s) do
    valof {}(B.$V) is
query {((B.$V).1), ((B.$V).2)} is
             {0, _} => output 0
             {_, 0} => output 0
             {1, 1} => output 1
         end : output query{ (((B.$V).s).1), (((B.$V).s).2)} is
                        {0, _} => output 0
                        {_, 0} => output 0
                        {1, 1} => output 1
                      end
    end
  end
end;

let min = fix(min_rec);
```

B.3 CDS0 algorithms used to compile PCF

We list the base environment of CDS0 dcds declarations and algorithms which are used to compile PCF programs. There are two versions of this base environment: one defines refinements of *bool* and *intlist*, and the other does not. We show the code for the refined version.

```
(* Constants that are part of the PCF environment *)
(* Automatically loaded in when user switches to  *)
```

```
(* PCF interpreter. *)

(* The basic types *)

let bool = dcds cell B values tt,ff end;
let int = dcds cell N values [..] end;

(* The refinement types *)

let true = dcds cell B values tt end;
let false = dcds cell B values ff end;
refine true, false;

(* The primitive operations *)

(* cond : ((bool * 'a) * 'a) -> 'a  *)
let cond =
  algo
    request $C do
      valof ((B.1).1) is
        tt: valof (($C.2).1) is
              $V: output $V
            end
        ff: valof ($C.2) is
              $W: output $W
            end
      end
    end
  end;

(* fst : ('a * 'b) -> 'a  *)
let fst = algo
    request $C do
        valof ($C.1) is
            $V: output $V
        end
    end
end;

(* snd : ('a * 'b) -> 'a  *)
let snd = algo
    request $C do
        valof ($C.2) is
            $W: output $W
        end
    end
end;

(* plus : (int * int) -> int  *)
let plus = algo
    request N do
        valof (N.1) is
            $V1: valof (N.2) is
```

```
                    $V2: output $V1 + $V2
                end
        end
    end
end;

let minus = algo
    request N do
        valof (N.1) is
            $V1: valof (N.2) is
                    $V2: output $V1 - $V2
                end
        end
    end
end;

let times = algo
    request N do
        valof (N.1) is
            $V1: valof (N.2) is
                    $V2: output $V1 * $V2
                end
        end
    end
end;

let div = algo
    request N do
        valof (N.1) is
            $V1: valof (N.2) is
                    $V2: output $V1 / $V2
                end
        end
    end
end;

(* equal : (int * int) -> bool  *)
let equal = algo
    request B do
        valof (N.1) is
            $V1: valof (N.2) is
                    $V2 with $V2 = $V1: output tt
                    $V2 with $V2 != $V1: output ff
                end
        end
    end
end;

let less = algo
    request B do
        valof (N.1) is
            $V1: valof (N.2) is
                    $V2 with $V2 > $V1: output tt
```

```
                    $V2 with $V2 <= $V1: output ff
                end
        end
    end
end;

let grtr = algo
    request B do
        valof (N.1) is
            $V1: valof (N.2) is
                    $V2 with $V2 < $V1: output tt
                    $V2 with $V2 >= $V1: output ff
                end
        end
    end
end;

let leq = algo
    request B do
        valof (N.1) is
            $V1: valof (N.2) is
                    $V2 with $V2 >= $V1: output tt
                    $V2 with $V2 < $V1: output ff
                end
        end
    end
end;

let geq = algo
    request B do
        valof (N.1) is
            $V1: valof (N.2) is
                    $V2 with $V2 <= $V1: output tt
                    $V2 with $V2 > $V1: output ff
                end
        end
    end
end;

(* and : (bool * bool) -> bool  *)
let land =
  algo
    request B do
      valof (B.1) is
        tt: valof (B.2) is
              tt: output tt
              ff: output ff
            end
        ff: output ff
      end
    end
  end;
```

```
let lor =
  algo
    request B do
      valof (B.1) is
        tt: output tt
        ff: valof (B.2) is
              tt: output tt
              ff: output ff
            end
      end
    end
  end;

(* Now things that are not explicitly in the language but are needed *)
(* in the translation to categorical combinators. *)

(* id : 'a -> 'a  *)
let id = algo
    request $C do
        valof $C is
            $V : output $V
        end
    end
end;

(* The empty environment *)
let emptyenv = {};

(* the "regular" fixpoint, Y : ('a -> 'a) -> 'a  *)
(*  Y = fix (fn f => fn x => x (f x))           *)
let Y = fix((curry(curry(uncurry(id) |
                   <snd, uncurry(id) | <snd|fst, snd>>))).emptyenv);

(* the "environment" fixpoint, Yenv : (env -> 'a -> 'a) -> env -> 'a  *)
let Yenv = curry(Y | uncurry(id));

(* Integer lists *)

letrec intlist = dcds
  cell EMPTY values true, false
  graft (int.l) access EMPTY = false
  graft (intlist.l) access EMPTY=false
end;

(* refined types *)

let empty_intlist = dcds
  cell EMPTY values true
end;

let one_intlist = dcds
  cell EMPTY values false
  cell (N.l) values [..] access EMPTY = false
```

```
  cell (EMPTY.l) values true access EMPTY = false
end;

local letrec partial_intlist = dcds
  cell (EMPTY.l) values true, false access EMPTY = false
  cell (N.l) values [..] access EMPTY = false
  graft (partial_intlist.l) access EMPTY = false
  end
in let many_intlist = dcds
  cell EMPTY values false
  cell (N.l) values [..] access EMPTY = false
  cell (EMPTY.l) values false access EMPTY = false
  cell ((N.l).l) values [..] access (EMPTY.l) = false
  graft (partial_intlist.l)
  end
end;

refine empty_intlist, one_intlist, many_intlist;

let nil = {EMPTY = true};

let null = algo
  request B do
    valof EMPTY is
      true : output tt
      false : output ff
    end
  end
end;

let cons = algo
  request EMPTY do
    output false
  end
  request (N.l) do
    valof (N.1) is
      $V : output $V
    end
  end
  request (EMPTY.l) do
    valof (EMPTY.2) is
      $B : output $B
    end
  end
  request (((EMPTY.$T).l).l) do
    from {((EMPTY.$T).2)=false} do
      valof (((EMPTY.$T).l).2) is
        $B : output $B
      end
    end
  end
  request (((N.$T).l).l) do
    from {((EMPTY.$T).2)=false} do
```

```
      valof (((N.$T).1).2) is
        $V : output $V
      end
    end
  end
end;

let hd = algo
  request N do
    valof EMPTY is
      false : valof (N.1) is
                $V : output $V
      end
    end
  end
end;

let tl = algo
  request (EMPTY.$T) do
    from {(EMPTY.$T)=false} do
      valof ((EMPTY.$T).1) is
        $B : output $B
      end
    end
  end
  request ((N.$T).1) do
    from {(EMPTY.$T)=false, ((EMPTY.$T).1)=false} do
      valof (((N.$T).1).1) is
        $V : output $V
      end
    end
  end
end;
```

B.4 Types for CDS0 algorithms in base environment

```
- cds0(refined);
-- Loading PCF constants.
Type bool defined.
Type int defined.
Type true defined.
Type false defined.
r: /\[((false * 'a) * 'b) -> 'b, ((true * 'c) * 'd) -> 'c]
 : ((bool * 'a) * 'a) -> 'a
Abbreviation "cond" defined.
r: ('a * 'b) -> 'a
 : ('a * 'b) -> 'a
Abbreviation "fst" defined.
r: ('a * 'b) -> 'b
 : ('a * 'b) -> 'b
Abbreviation "snd" defined.
r: (int * int) -> int
```

```
 : (int * int) -> int
Abbreviation "plus" defined.
r: (int * int) -> int
 : (int * int) -> int
Abbreviation "minus" defined.
r: (int * int) -> int
 : (int * int) -> int
Abbreviation "times" defined.
r: (int * int) -> int
 : (int * int) -> int
Abbreviation "div" defined.
r: (int * int) -> bool
 : (int * int) -> bool
Abbreviation "equal" defined.
r: (int * int) -> bool
 : (int * int) -> bool
Abbreviation "less" defined.
r: (int * int) -> bool
 : (int * int) -> bool
Abbreviation "grtr" defined.
r: (int * int) -> bool
 : (int * int) -> bool
Abbreviation "leq" defined.
r: (int * int) -> bool
 : (int * int) -> bool
Abbreviation "geq" defined.
r: /\[(false * bool) -> false, (true * true) -> true, (true * false) -> false]
 : (bool * bool) -> bool
Abbreviation "land" defined.
r: /\[(false * false) -> false, (true * bool) -> true, (false * true) -> true]
 : (bool * bool) -> bool
Abbreviation "lor" defined.
r: 'a -> 'a
 : 'a -> 'a
Abbreviation "id" defined.
r: 'a
 : 'a
Abbreviation "emptyenv" defined.
r: ('a -> 'a) -> 'a
 : ('a -> 'a) -> 'a
Abbreviation "Y" defined.
r: ('a -> 'b -> 'b) -> 'a -> 'b
 : ('a -> 'b -> 'b) -> 'a -> 'b
Abbreviation "Yenv" defined.
Type intlist defined.
Type empty_intlist defined.
Type one_intlist defined.
Type many_intlist defined.
r: empty_intlist
 : intlist
Abbreviation "nil" defined.
r: /\[one_intlist -> false, empty_intlist -> true, many_intlist -> false]
 : intlist -> bool
```

```
Abbreviation "null" defined.
r: /\[(int * one_intlist) -> many_intlist, (int * many_intlist) -> many_intlist,
      (int * empty_intlist) -> one_intlist]
 : (int * intlist) -> intlist
Abbreviation "cons" defined.
r: /\[one_intlist -> int, many_intlist -> int]
 : intlist -> int
Abbreviation "hd" defined.
r: /\[many_intlist -> intlist, one_intlist -> empty_intlist]
 : intlist -> intlist
Abbreviation "tl" defined.

CDS0 version 1.1 --- June 11, 1997
#
```

B.5 AND_TASTER

```
# let AND_TASTER =
  algo
    request WHICH_AND do
      valof {}B is
        output tt: output IS_NOT_AN_AND
        output ff: output IS_NOT_AN_AND
        valof (B.1):
          valof {(B.1)=tt}B is
            output tt: output IS_NOT_AN_AND
            output ff: output IS_NOT_AN_AND
            valof (B.2):
              valof {(B.1)=tt,(B.2)=tt}B is
                output ff: output IS_NOT_AN_AND
                output tt:
                  valof {(B.1)=tt,(B.2)=ff}B is
                    output tt: output IS_NOT_AN_AND
                    output ff:
                      valof {(B.1)=ff}B is
                        output tt: output IS_NOT_AN_AND
                        output ff: output IS_LEFT_AND
                        valof (B.2):
                          valof {(B.1)=ff,(B.2)=tt}B is
                            output tt: output IS_NOT_AN_AND
                            output ff:
                              valof {(B.1)=ff,(B.2)=ff}B is
                                output tt: output IS_NOT_AN_AND
                                output ff: output IS_LEFT_STRICT_AND
                              end
                          end
                      end
                  end
                end
          end
valof (B.2):
          valof {(B.2)=tt}B is
```

```
            output tt: output IS_NOT_AN_AND
            output ff: output IS_NOT_AN_AND
            valof (B.1):
              valof {(B.2)=tt,(B.1)=tt}B is
                output ff: output IS_NOT_AN_AND
                output tt:
                  valof {(B.2)=tt,(B.1)=ff}B is
                    output tt: output IS_NOT_AN_AND
                    output ff:
                      valof {(B.2)=ff}B is
                        output tt: output IS_NOT_AN_AND
                        output ff: output IS_RIGHT_AND
                        valof (B.1):
                          valof {(B.2)=ff,(B.1)=tt}B is
                            output tt: output IS_NOT_AN_AND
                            output ff:
                              valof {(B.2)=ff,(B.1)=ff}B is
                                output tt: output IS_NOT_AN_AND
                                output ff: output IS_RIGHT_STRICT_AND
                              end
                          end
                      end
                  end
                end
          end
      end
    end
  end;

r: /\[/\[(false * false) -> false, (true * true) -> true,
         (true * false) -> false, (false * true) -> false] -> is_and_type,
      ((bool * bool) -> true) -> is_not_and_type,
      /\[(false * false) -> true, (true * true) -> true,
         (true * false) -> false, (false * true) -> false] -> is_not_and_type,
     ((bool * bool) -> false) -> is_not_and_type,
     /\[(true * false) -> true, (true * true) -> true,
        (false * true) -> false] -> is_not_and_type,
     ((true * bool) -> true) -> is_not_and_type,
     /\[(bool * false) -> false, (true * true) -> true,
        (false * true) -> false] -> is_and_type,
     ((true * bool) -> false) -> is_not_and_type,
     /\[(bool * false) -> true, (true * true) -> true,
        (false * true) -> false] -> is_not_and_type,
     ((true * true) -> false) -> is_not_and_type,
     /\[(false * true) -> true, (true * true) -> true] -> is_not_and_type,
     /\[(true * false) -> true, (true * true) -> true] -> is_not_and_type,
     ((bool * true) -> false) -> is_not_and_type,
     /\[(false * bool) -> true, (true * true) -> true,
        (true * false) -> false] -> is_not_and_type,
     ((bool * true) -> true) -> is_not_and_type,
     /\[(false * bool) -> false, (true * true) -> true,
        (true * false) -> false] -> is_and_type,
     /\[(false * false) -> false, (true * true) -> true,
```

```
       (false * true) -> false, (true * false) -> false] -> is_and_type,
    /\[(false * true) -> true, (true * true) -> true,
    (true * false) -> false] -> is_not_and_type,
    /\[(false * false) -> true, (true * true) -> true,
       (false * true) -> false, (true * false) -> false] -> is_not_and_type]
 : ((bool * bool) -> bool) -> and_type
Abbreviation "AND_TASTER" defined.
```

Appendix C

CDS0 and PCF Syntax

We present the syntax from our implementation of CDS0 and PCF in form similar to ML-Yacc input (semantic actions being omitted). Names in capitals are terminals, all lower case are non-terminals. Where there is the possibility for confusion, we have placed single quotes around a symbol.

C.1 CDS0 syntax

```
prog :
     | expr
     | command
     | dcds_decla

  (* Expressions--begin *)
expr : { event_list }
     | algo_decl
     | CURRY ( expr )
     | UNCURRY ( expr )
     | expr `|' expr
     | expr . expr
     | < expr , expr >
     | ( expr , expr )
     | FIX ( expr )
     | ( expr )
     | ID
  (* Expressions--end *)

  (* Commands--begin *)
command : LET ID = expr
        | PRINT ID
        | LOAD FILE
        | LOADECHO FILE
        | TRACE ON
        | TRACE OFF
        | TIMER ON
        | TIMER OFF
        | TYPING ON
        | TYPING OFF
```

```
        | SHOW INTEGER ID
        | SHOW MORE INTEGER ID
        | HIERARCHY FILE
        | ENV
        | PCF
  (* Commands--end *)

  (* Dcds declaration--begin *)
dcds_decla : LETREC dcds_decl
           | LET dcds_decl

dcds_decl  : ID = DCDS component END

component  :
           | CELL cell_name VALUES value_list access_list component
           | GRAFT cell_name access_list component

cell_name  : ID
           | VAR
           | { event_list } cell_name
           | ( cell_name . tag )

tag : ID
    | arexpr
    | interval

value : ID
      | VALOF cell_name
      | OUTPUT value
      | arexpr
      | ( value . value )
      | VAR WITH boolexp
      | interval

interval : [ . . ]
         | [ int . . ]
         | [ . . int ]
         | [ int . . int ]

int : INTEGER
    | ~ INTEGER

value_list : value
           | value , value_list

access_list :
            | ACCESS enabling

enabling : event_list
         | event_list OR enabling

event_list :
           | event
```

```
          | event , event_list

event : cell_name = value
  (* Dcds declaration--end *)

  (* Arithmetic expression--begin *)
arexpr : int
       | VAR
       | ~ VAR
       | ~ ( arexpr )
       | arexpr PLUS arexpr
       | arexpr SUB arexpr
       | arexpr MULT arexpr
       | arexpr DIV arexpr
       | ( arexpr )
  (* Arithmetic expression--end *)

  (* Boolean expressions--begin *)
boolexp : arexpr > arexpr
        | arexpr >= arexpr
        | arexpr < arexpr
        | arexpr <= arexpr
        | value = value
        | value != value
        | boolexp OR boolexp
        | boolexp AND boolexp
        | ( boolexp )
  (* Boolean expressions--end *)

  (* Algorithm declaration--begin *)
algo_decl : ALGO body_list END

body_list :
          | body_list body

body : REQUEST ext_cell_name DO instruction END

ext_cell_name : cell_name
              | cell_name WITH boolexp

instruction : OUTPUT value
            | VALOF cell_name IS query_list END
            | from_do_list
            | OMEGA

from_do_list : from_do
             | from_do_list from_do

from_do : FROM { event_list } DO instruction END

query : value ‘:’ instruction

query_list :
```

```
          | query_list query
  (* Algorithm declaration--end *)
```

C.2 PCF syntax

```
program :
     | expr
     | VAL ID = expr
     | LOAD FILE
     | PRINT ID
     | QUIT

expr : TRUE
     | FALSE
     | int
     | ID
     | expr expr
     | FN ID => expr
     | LET ID = expr IN expr END
     | LETREC ID = expr IN expr END
     | bop
     | IF expr THEN expr ELSE expr
     | ( expr , expr )
     | FST expr
     | SND expr
     | expr :: expr
     | HD expr
     | TL expr
     | [ ]
     | NULL expr
     | ( expr )

int : INTEGER
    | ~ INTEGER

bop : expr + expr
    | expr - expr
    | expr * expr
    | expr / expr
    | expr = expr
    | expr < expr
    | expr > expr
    | expr <= expr
    | expr >= expr
    | expr AND expr
    | expr OR expr
```

Bibliography

[1] S. Abramsky, Computational interpretations of linear logic, in: *Theoretical Computer Science* 111 (1993), 3–57.

[2] S. Abramsky, R. Jagadeesan, P. Malacaria, Full abstraction for PCF (extended abstract), in: *Theoretical Aspects of Computer Software, Sendai, Japan, 1994*, (Springer LNCS 789, 1994), 1–15.

[3] M. Ajtai, J. Komlós, E. Szemerédi, An $O(n \lg n)$ sorting network, in: *Proc. ACM Symposium on the Theory of Computation, 1983*, 1–9.

[4] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, An intensional parallel processing language for applications programming, Technical Report SRI-CSL-89-1, SRI International, 1989.

[5] G. Berry, Programming with concrete data structures and sequential algorithms, in: *Proc. ACM Conf. on Functional Programming Languages And Computer Architecture, Wentworth-by-the-Sea, 1981*, 49-57.

[6] G. Berry and P.-L. Curien, Sequential algorithms on concrete data structures, *Theoretical Computer Science* 20 (1985), 265-321.

[7] G. Berry and P.-L. Curien, The kernel of the applicative language CDS: Theory and practice, in: M. Nivat and J.C. Reynolds, eds., *Algebraic Methods in Semantics* (Cambridge University Press, 1985) 35-87.

[8] G. Berry, P.-L. Curien, J.-J. Lévy, Full abstraction for sequential languages: the state of the art, same source as [7], 89-132.

[9] G.E. Blelloch and J. Greiner, A parallel complexity model for functional languages, in: *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, 1995*.

[10] R.B. Boppana and M. Sipser, The complexity of finite functions, in: J. van Leeuwen ed., *Handbook of Theoretical Computer Science, Vol. A* (Elsevier, 1990), 749–804.

[11] S. Brookes and S. Geva, Computational Comonads and Intensional Semantics, Technical Report CMU-CS-91-190, Carnegie Mellon, 1991.

[12] S. Brookes and S. Geva, Towards a theory of parallel algorithms on concrete data structures, *Theoretical Computer Science* 101 (1992) 177-221.

[13] S. Brookes and D. Dancanet, Sequential algorithms, deterministic parallelism, and intensional expressiveness, in: *Proc. ACM Symposium on Principles of Programming Languages, 1995*, 13–24.

[14] R. Cartwright, P.-L. Curien, M. Felleisen, Fully abstract semantics for observably sequential languages, to appear in *Information and Computation*.

[15] R. Cartwright and M. Fagan, Soft typing, in: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, 1991*, 278–292.

[16] G. Castagna, G. Ghelli, and G. Longo, A calculus for overloaded functions with subtyping, in: *Proc. ACM Conf. on Lisp and Functional Programming, 1992*, 182–192.

[17] P. Clote, A sequential programming language for the parallel complexity class NC, Boston College, Technical report BCCS-88-07, 1988.

[18] A. Cobham, The intrinsic computational difficulty of functions, in: Y. Bar-Hillel, ed., *Logic, Methodology and Philosophy of Science II, Jerusalem 1964* (North-Holland, 1965), 24-30.

[19] L. Colson, About primitive recursive algorithms, in: G. Ausiello et al. eds., *Proc. 16th International Colloquium on Automata, Languages and Programming* (Springer-Verlag LNCS 372, 1989), 194-206.

[20] L. Colson, *Représentation intentionelle d'algorithmes dans les systémes fonctionelles: une étude de cas*, Thése de Doctorat, Université Paris VII (1991).

[21] M. Coppo and P. Giannini, A complete type inference algorithm for simple intersection types, in: *Proc. 17th Colloq. on Trees and Algebra in Programming, 1992*, 102–123.

[22] T. Coquand, Une preuve directe du Théorème d'Ultime Obstination, *Comptes Rendus de l'Academie des Sciences*, March 1992.

[23] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms* (MIT Press, 1990).

[24] B. Courcelle, Fundamental properties of infinite trees, in: *Theoretical Computer Science* 25 (1983), 95–109.

[25] G. Cousineau, P.-L. Curien, M. Mauny, The categorical abstract machine, in: *Science of Computer Programming* 8 (1987) 173–202.

[26] P.-L. Curien, *Categorical Combinators, Sequential Algorithms, and Functional Programming* (Birkhäuser, 1993).

[27] D. Dancanet and S. Brookes, Programming language expressiveness and circuit complexity, in: *Internat. Conf. on the Mathematical Foundations of Programming Semantics, 1996*.

[28] D. Dancanet, CDS0 User's Guide (version 1.1).

[29] R. David, The Inf function in the system F, manuscript.

[30] M. Devin, *Le Langage CDS: Description, Implémentation, Compilation*, Thèse Docteur Ingénieur, Université Paris VII (1984), Rapport LITP 85-13.

[31] A.A. Faustini and W.W. Wadge, Intensional programming, in: J.C. Boudreaux et al. eds., *The Role of Language in Problem Solving 2*, (North-Holland, 1987), 119-132.

[32] M. Felleisen, On the expressive power of programming languages, *Science of Computer Programming* 17 (1991) 35–75.

[33] A. Ferguson and J. Hughes, Fast abstract interpretation using sequential algorithms, in: *Proc. Padova Workshop on Static Analysis, 1993*.

[34] M.J. Fischer, The consensus problem in unreliable distributed systems (a brief survey), in: M. Karpinski, ed., *Proc. Internat. Conf. on Foundations of Computation Theory* (Springer, 1983) 127–140.

[35] T. Freeman and F. Pfenning, Refinement types for ML, in: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, 1991*.

[36] T. Freeman, *Refinement types for ML*, Doctoral Thesis, Carnegie Mellon University, Technical Report CMU-CS-94-110, March 1994.

[37] Y.-C. Fuh and P. Mishra, Type inference with subtypes, in: *Theoretical Computer Science* 73 (1990), 155–175.

[38] J.-Y. Girard, *Proof Theory and Logical Complexity I* (Bibliopolis, 1987).

[39] J.-Y. Girard, Y. Lafont, P. Taylor, *Proofs and Types* (Cambridge Tracts in Theoretical Computer Science 7, 1990).

[40] J.-Y. Girard, A. Scedrov, P.J. Scott, Bounded linear logic: a modular approach to polynomial-time computability, in: *Theoretical Computer Science* 97 (1992) 1–66.

[41] J. Greiner and G.E. Blelloch, A provably efficient fully speculative parallel implementation, in: *Proc. ACM Symposium on Principles of Programming Languages, 1996*.

[42] C.A. Gunter, *Semantics of Programming Languages* (MIT Press, 1992).

[43] C.A. Gunter and J.C. Mitchell, *Theoretical Aspects of Object-Oriented Programming* (MIT Press, 1994).

[44] D.J. Gurr, *Semantic Frameworks for Complexity*, Doctoral Thesis, University of Edinburgh, Technical Report ECS-LFCS-91-130, January 1991.

[45] C. Hankin and S. Hunt, Approximate fixed points in abstract interpretation, in: *European Symposium on Programming*, (Springer LNCS 582, 1992).

[46] J.R. Hindley, Types with intersection: An introduction, in: *Formal Aspects of Computing* 4 (1992), 470–486.

[47] C.A.R. Hoare, Communicating Sequential Processes, in: *Communications of the ACM* 21 (8) 1978, 666–677.

[48] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, 1979).

[49] P. Hudak and S. Anderson, Pomset interpretations of parallel functional programs, in: *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, 1987*, 234–256.

[50] J. Hughes and A. Ferguson, A loop-detecting interpreter for lazy, higher-order programs, in: *Proc. Glasgow Workshop on Functional Languages, 1992*.

[51] J. Hughes, S. Hunt, C. Runciman, Higher-order functions as decision trees: Taming a space monster, 1994, manuscript.

[52] J.M.E. Hyland and C.-H.L. Ong, On full abstraction for PCF: I, II, and III, manuscript.

[53] N.D. Jones, Constant time factors do matter, in: *Proc. ACM Symposium on the Theory of Computation, 1993*, 602–611.

[54] N.D. Jones, Computability and complexity from a programming perspective, in: *Proc. Mathematical Foundations of Programming Semantics, 1995.*

[55] G. Kahn and D.B. MacQueen, Coroutines and networks of parallel processes, in: *Information Processing 77* (North-Holland, 1977) 993-998.

[56] G. Kahn and G.D. Plotkin, Concrete Domains, in: *Theoretical Computer Science* 121(1993). Earlier available in French as: Domaines Concrets, IRIA Report 336, 1978.

[57] S.C. Kleene, *Introduction to Metamathematics* (North-Holland, 1952).

[58] L. Lamport, Using time instead of timeout for fault-tolerant distributed systems, in: *ACM Trans. on Programming Languages and Systems* 6(2) 1984, 254–280.

[59] D. Leivant and J.-Y. Marion, Lambda calculus characterizations of poly-time, in: *Typed Lambda Calculi and Applications, 1993*, 274–288.

[60] D. Le Metayer, Mechanical Analysis of Program Complexity, in: *ACM SIGPLAN Symposium on Language Issues in Programming Environments, Seattle, 1985* 69-73.

[61] M. Mauny and A. Suarez, Implementing functional languages in the categorical abstract machine, in: *Proc. of the ACM Conf. on Lisp and Functional Programming, 1986*, 266–278.

[62] J. McCarthy, A basis for a mathematical theory of computation, in: Braffort and Hirschberg, eds., *Computer Programming and Formal Systems* (North-Holland, 1963), 33–70.

[63] R. Milner, Fully abstract models of typed λ-calculi, in: *Theoretical Computer Science* 4(1977), 1–22.

[64] Y. Moschovakis, Abstract recursion as a foundation for the theory of algorithms, in: M.M. Richter et al. eds., *Computation and Proof Theory* (Springer-Verlag LNM 1104, 1984), 289–364.

[65] Y. Moschovakis, The Formal Language of Recursion, *The Journal of Symbolic Logic*, vol. 54, 1989, 1216–52.

[66] Y. Moschovakis, A mathematical modeling of pure, recursive algorithms, in: A. Meyer and M.A. Taitslin eds., *Logic at Botik '89: Symposium on Logical Foundations of Computer Science* (Springer-Verlag, 1989), 208–29.

[67] H. Nickau, *Hereditarily sequential functionals: A game-theoretic approach to sequentiality*, Doctoral Thesis, Universität Siegen (Shaker Verlag, 1996).

[68] P. Panangaden and V. Shanbhogue, On the expressive power of indeterminate network primitives, Cornell University, Technical Report TR 87-891, 1987.

[69] B.C. Pierce, *Programming with intersection types and bounded polymorphism*, Doctoral Thesis, Carnegie Mellon University, Technical Report CMU-CS-91-205, December 1991.

[70] G.D. Plotkin, LCF considered as a programming language, in: *Theoretical Computer Science* 5(1977), 223-56.

[71] G.D. Plotkin, A structural approach to operational semantics, University of Aarhus, Technical Report DAIMI FN-19, 1981.

[72] R. Raz and A. Wigderson, Monotone circuits for matching require linear depth, in: *ACM Symposium on the Theory of Computation, 1990*, 287–292.

[73] J. Reppy, CML: A Higher-Order Concurrent Language, revised version of paper presented at *SIGPLAN Conf. on Programming Language Design and Implementation, 1991*, 1993.

[74] J.C. Reynolds, Preliminary design of the programming language Forsythe, Carnegie Mellon University, Technical Report CMU-CS-88-159, June 1988.

[75] P. Roe, Calculating lenient programs' performance, in: S.L. Peyton-Jones, G. Hutton, C. Kehler Holst, eds., *Functional Programming, Glasgow 1990* (Springer, 1990) 227–236.

[76] H. Rogers, Jr. , *Theory of Recursive Functions and Effective Computability* (MIT Press, 1987).

[77] M. Rosendahl, Automatic complexity analysis, in: *Proc. Functional Programming Languages and Computer Architecture, 1989*, 144-156.

[78] D. Sands, Complexity analysis for a lazy higher-order language, in: *Proc. Third European Symposium on Programming, 1990*, 361-76.

[79] D. Sands, Time analysis, cost equivalence and program refinement, in: *Proc. Foundations of Software Technology and Theoretical Computer Science, New Delhi, 1991*, 25-39.

[80] D.A. Schmidt, *Denotational Semantics* (Allyn and Bacon, 1986).

[81] A. Stoughton, Interdefinability of parallel operations in PCF, *Theoretical Computer Science* 79 (1991) 357-8.

[82] S. Sur and W. Böhm, Functional, I-structure, and M-structure implementations of NAS benchmark FT, in: *Parallel Architectures and Compilation Techniques, 1994*, 47–56.

[83] C. Talcott, Rum: An intensional theory of function and control abstractions, in: *Workshop on Foundations of Logic and Functional Programming, Trento, 1986* (Springer, 1988) 3-44.

[84] É. Tardos, The gap between monotone and non-monotone circuit complexity is exponential, *Combinatorica* 7 (4) (1987), 141–2.

[85] P. Wadler, Strictness analysis aids time analysis, in: *Proc. Symposium on Principles of Programming Languages, San Diego, 1988*, 119-32.

[86] B. Wegbreit, Mechanical Program Analysis, *Communications of the ACM* 18(9) 1975, 528-39.

[87] I. Wegener, *The Complexity of Boolean Functions* (Wiley, 1987).

[88] J. Young and P. Hudak, Finding fixed points on function spaces, Technical Report YALEU/DCS/RR-505, Dept. of Computer Science, Yale University, December 1986.

[89] W. Zimmermann, Automatic worst case complexity analysis of parallel programs, Technical Report ICSI 90-066, International Computer Science Institute, 1990.

[90] W. Zimmermann, The automatic worst case analysis of parallel programs: simple parallel sorting and algorithms on graphs, Technical Report ICSI 91-045, International Computer Science Institute, 1991.