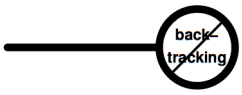
Linear and Affine Typing of Continuation-Passing Style



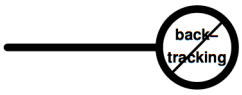
Joshua James Berdine

Submitted for the degree of Doctor of Philosophy

Queen Mary, University of London

2004

Linear and Affine Typing of Continuation-Passing Style



Joshua James Berdine

Abstract

In this dissertation we show that linear and affine type systems for continuation-passing style support correct and tight refinements of standard continuation semantics. In particular, a wide variety of control constructs admit typing disciplines which ensure linear or affine use of the control context in their continuation semantics.

This refinement of standard continuation semantics using restricted types is an exploitation of the stylized use of continuations many control behaviors exhibit. Continuations are the raw material of control and can be used to explain a wide variety of control behaviors, including calling/returning (procedures), raising/handling (exceptions), jumping/labeling (`goto` and labels), process switching (coroutines), backtracking (`amb` and `fail`), and capturing/invoking first-class continuations (`call/cc`, or `callcc` and `throw`). However, in all but the last case, continuations are not themselves intrinsic to the control construct, instead they are "behind the scenes," implementing the control construct. In other words, except for first-class continuations, each control behavior is simply an idiom of continuation usage, and hence the continuations are used in a stylized fashion. Linear or affine use of control contexts; by which we mean, roughly, that control contexts cannot be duplicated or, in the linear case, discarded; captures this stylized usage in all the above-mentioned cases bar backtracking. We also investigate several cases where even storable labels (plus `goto`) admit restricted interpretations.

Submitted for the degree of Doctor of Philosophy

Queen Mary, University of London

2004

Contents

List of Figures

Acknowledgements

Some of the research presented in this dissertation is joint work with Peter O'Hearn, Uday Reddy, and Hayo Thielecke [BOT00, BORT00, BORT02, BOT02]. O'Hearn and Thielecke had the basic idea for the soundness of the treatment of λ-calculus in Chapter 2 and Chapter 3 before I got involved. (For what it is worth, the same idea is also the basis of soundness of the simple command language in Chapter 6.) The basic idea for soundness of exceptions in Chapter 4 came out of discussions they had with Reddy. Soundness of forward jumps in Chapter 7 is largely based on this same idea. Completeness for λ-calculus can be seen as a logical reconstruction of (part of) Sabry and Felleisen's work [SF93]. The core of completeness for exceptions, the direct-style transformation in Section 4.5.2, is due to Thielecke. In addition to all the technical formulations and results, I am responsible for the distinction between continuations and control contexts, the use of delimited continuations, and the methods of handling the store (Chapter 5 and from Chapter 8 to the end).

I am intensely grateful to Peter O'Hearn and Hayo Thielecke, my advisors, for their insight and talent, for exciting and inspiring me, and for coping with my often illogical rants. More thanks to Peter for first showing me how cool this computer science stuff is in Freshman year and sucking me into the programming languages world (and then dragging me across the Atlantic). In addition to Peter and Hayo, I also want to thank Per Brinch Hansen, John Reynolds, and Jim Royer for dramatically impacting who I am as a computer scientist through discussions and their work. In addition to those already mentioned, I and this work have benefitted greatly from discussions with Richard Bornat, Cristiano Calcagno, Olivier Danvy, and Paul Blain Levy.

This research was financially supported by the Overseas Research Students Awards scheme and the Department of Computer Science, Queen Mary, University of London.

Finally, and originally, I thank my family and friends for getting me here as I am, putting up with me, being ignored, you know.

Chapter 1

Introduction

1.1 Note to the Reader

Note that reading this chapter is important for the reader familiar with continuations and continuation-passing style. The continuations literature is full of ambiguous or multiply defined terms, and this chapter sets our point of view and definitions of terms which the later discussion will assume. Utter confusion is quite possible later if, for instance, a different notion of "continuation" than we present here is assumed.

It is assumed that the reader has some working understanding of continuations, continuation-passing style, and continuation semantics. Thielecke's logic column [Thi99a] provides a brief and conceptual introduction to these topics. A more complete introduction is given in Friedman, Wand, and Haynes' text [FWH92, FWH01]. Reynolds focuses more on continuation semantics than on continuation-passing style in his text [Rey98b], and considers a wider variety of instances of the ideas. Steele and Sussman's original reports [SS98, SS76, SS78, SS79, SS80, Ste76, Ste77a, Ste77b, Ste78, Ste80] and Appel's text [App92] are also rich in intuitions.

Knowing some basic type theory will help. Pierce's text [Pie02] covers far more than we will need. Some understanding of linear logic might help, though I have attempted to make no assumptions in this regard, and our use of linear logic will be very specialized.

We first, in Section 1.2, dive hard into some continuation programming to get a feel for the sort of control behavior we will later be concerned with restricting. This could be rough going for those not used to the insanity that is continuation programming, but understanding it fully is not a prerequisite for understanding later chapters, and it may still provide some appropriate context. After that, in Section 1.3, we will much more calmly introduce the basic ideas of our approach. Then, in Section 1.4, we place this work and head off potential confusions for experts. Hence, it is rather rougher going than other sections on readers not (yet) embroiled in the world of continuations, who may be better off skimming this section and referring back as necessary. Section 1.5 details a few motivations for this work, and Section 1.6 summarizes the main conceptual contributions.

1.2 Using a Continuation Twice

Our focus will be on prohibiting certain control behaviors through the use of restricted type systems for continuation-passing style. So to set the stage, we first give an example of the sort of control behavior we will be concerned with. Consider the following call-by-value λ-calculus term:

$$\lambda z.\, \lambda x.\, x$$

Figure 1.1 `return/cc` and `backtrack?` in Scheme

```
(define call/cc call-with-current-continuation)

(define return/cc
  (lambda ()
    (call/cc (lambda (k) k))))

(define backtrack?
  (lambda (testee)
    (let ((ratchet (list 'anything #f #t)))
      (let ((f (testee)))
        (begin
          (set! ratchet (cdr ratchet))
          (f (lambda (x) 'anything))
          (car ratchet))))))
```

This is almost as trivial as λ-terms come: it returns a procedure which when called returns the identity procedure. The continuation-passing style (CPS) version of this term is

$$\lambda r.\, r\, \lambda k.\, \lambda z.\, k\, \lambda h.\, \lambda x.\, h\, x$$

While more explicit and verbose, this term is simple enough: it accepts a (toplevel) continuation and invokes it with the CPS version of the value $\lambda z.\, \lambda x.\, x$, which accepts a (return) continuation, k, and an argument, and invokes k with the CPS version of the value $\lambda x.\, x$, which accepts a continuation, h, and argument, x, and returns x to h.

But now consider the CPS term:

$$\lambda r.\, r\, \lambda k.\, \lambda z.\, k\, \lambda h.\, \lambda x.\, \boxed{k}\, x \tag{1.1}$$

We have simply changed the invocation of h to invoke k (boxed), which completely changes the meaning. In direct (that is, not continuation-passing) style, (1.1) is

$$\texttt{return/cc} \stackrel{\text{def}}{=} \lambda z.\, \texttt{call/cc}\, \lambda k.\, k$$

This procedure, instead of doing something as thoroughly benign as returning the identity procedure, returns the current continuation (wrapped into a procedure, as `call/cc` does) of the site from where it is called. So, calling this returned procedure will cause the computation to backtrack, and the call of `return/cc` will return again. More concretely, `return/cc` in Scheme along with Thielecke's `backtrack?`[1] [Thi99b] are shown in Figure 1.1. Now, evaluating

```
(backtrack? return/cc)
```

yields

```
#t
```

Tracing how this transpires, the first interesting point is when `(testee)` is called. Since `testee` is bound to `return/cc`, `f` is bound to a continuation which corresponds to

[1] Actually, we use a slight variant since the arguments applied to the procedures returned by `return/cc` must be procedures, rather than anything at all. This is understandable when one observes that the type of `return/cc` is $A \to \mu B.\, B \to \mathbf{R}$.

```
(lambda (v)
  (let ((f v))
    (begin
      (set! ratchet (cdr ratchet))
      (f (lambda (x) 'anything))
      (car ratchet))))
```

Then the ratchet is advanced once, and `f` is called. This returns `(lambda (x) 'anything)` to the continuation bound to `f`, at which point execution proceeds by evaluating

```
(let ((f (lambda (x) 'anything)))
  (begin
    (set! ratchet (cdr ratchet))
    (f (lambda (x) 'anything))
    (car ratchet)))
```

Then the ratchet is advanced once more, `f` is called, `'anything` is returned, and finally the ratchet is observed.

The point to note here is that `return/cc` is a procedure which, when called once, returns (to its first call site) twice. In this way, it "uses" its return continuation twice. Backtracking is accomplished by remembering the return continuation when returning the first time. Here, the current continuation is remembered simply by returning it, although other mechanisms are equally possible.

1.3 Constraining CPS: Linear Use of Control Contexts

What the example in the previous section demonstrates is that the power to express `call/cc` which is present in CPS is considerable. Adding `call/cc` to a programming language completely changes the character of the potential control-flow which programs may exhibit. Hence, for languages without constructs such as `call/cc`, translating into CPS causes a drastic amount of information loss. We aim to explore the conceptual issues surrounding a more constrained CPS into which languages with less expressive control constructs can be translated without this loss of information.

The starting point is the observation in the previous section (made previously by Thielecke [Thi99b]) that the key to the extreme expressiveness `call/cc` provides is the ability to invoke a continuation more than once. So we want to constrain CPS in such a way that continuations cannot be invoked more than once. Equipping CPS with a linear type system and interpreting programming languages such that continuations are used linearly—meaning, roughly, that continuations are neither duplicated nor discarded—allows us to capture this intuition.

The basic idea can be illustrated by considering the type used to interpret untyped call-by-value λ-calculus. As Scott [Sco70] gave a typed explanation of untyped λ-calculus using the recursive type

$$\mu D.\, D \to D$$

a standard CPS interpretation uses the recursive type

$$\mu D.\, (D \to \mathbf{R}) \to D \to \mathbf{R} \tag{1.2}$$

where $\mathbf{R}$ is a type of results.

This domain indicates that the CPS version of a procedure first accepts a return continuation (of type $D \to \mathbf{R}$), then accepts an argument (of type D), and then runs (produces a final result, of type $\mathbf{R}$). As the parameter of an ordinary λ-calculus function, there are no constraints on how the return continuation is used. But in λ-calculus, once a procedure returns, it will not

return again until called again, and it must return. Hence, the use of continuations by procedure calling/returning is very stylized, and can be captured by restricting the principal $(\cdot) \to (\cdot)$ (the type of ordinary functions) in (1.2) to $(\cdot) \multimap (\cdot)$ (the type of linear functions), yielding

$$\mu D.\, (D \to \mathbf{R}) \multimap D \to \mathbf{R} \tag{1.3}$$

Due to the stylized use of continuations procedures make, this type is expressive enough to interpret all of λ-calculus. Furthermore, control behavior beyond that expressible with procedures—such as that exhibited by backtracking programs like `return/cc`—can be interpreted with (1.2) but not with (1.3) since, as we later discuss, `call/cc` duplicates the current continuation and is ruled out by this more restrictive typing. Indeed, this restricted typing disallows all CPS terms which do not correspond to an actual program in the source language. More concretely, recall `return/cc` in CPS:

$$\lambda r.\, r\, \lambda k.\, \lambda z.\, \boxed{k}\, \lambda h.\, \lambda x.\, \boxed{k}\, x$$

and note the two occurrences of k (boxed). Attempting to give this term type (1.3) fails since the continuation k is duplicated: one copy is wrapped into a procedure and passed to another copy.[2]

While considering only procedures is sufficient to demonstrate the basic idea, there is an important conceptual issue lurking which procedures alone do not reveal. Later (Chapter 4) we will present a language with a simple exception handling mechanism. There is only one sort of exceptions, and in direct style the language could be interpreted, following Moggi, using the type

$$\mu D.\, D \to (D + \mathbf{E})$$

where $\mathbf{E}$ is the type of exceptions. We give a CPS interpretation using the type

$$\mu D.\, (D \to \mathbf{R}) \,\&\, (D \to \mathbf{R}) \multimap D \to R$$

where $\&$ is the linear additive product, and we have taken $\mathbf{E} = D$. The typing of $\&$ allows one or the other of the components of a $\&$-pair to be used, but not both. The point to note at this time is that it is the $\&$-pair of type $(D \to \mathbf{R}) \,\&\, (D \to \mathbf{R})$, rather than a continuation of type $D \to \mathbf{R}$, which is used linearly in this interpretation. These $\&$-pairs can be thought of as "semantic" continuations,[3] since they represent an abstraction of the effect of the rest of the computation. But instead of having two crucially different sorts of continuations around, we reserve the term continuation for "destination of a jump" continuations, having type $T \to \mathbf{R}$, and use the term control context for "rest of the computation" continuations. This same distinction is made in Standard ML of New Jersey where (in our terminology) `callcc` captures control contexts, including the current exception handler, while `capture` captures only continuations, which when invoked use the exception handler of the invoker. So in general, the restriction on CPS which we want to make is not that continuations are used linearly, but that control contexts are used linearly. Later, in Section 5.3, we will argue much more strongly that this distinction is indispensable by demonstrating how restricting the use of continuations but not control contexts fails to prohibit `call/cc`.

1.4 Vantage Point and Plan of Attack

Having briefly summarized our problem and approach, we now step back and fill in our starting point and perspective in some more detail and generality, and then describe the generic plan of

[2] This explanation of *why* the typing fails is not entirely accurate, but will do for now; and this issue is handled accurately later in Section 3.6.

[3] Additionally, $(D \to \mathbf{R}) \,\&\, (D \to \mathbf{R})$ is isomorphic to $(D + D) \to \mathbf{R}$, which looks more like a type of continuations.

which the following chapters are instances. The primary characteristic of our vantage point is the definitions of *continuation*, *control context*, *control environment*, and *control state*. While still informal, the definitions we state identify *which* of the informal definitions in the literature we are taking, and form the basis of all later conceptual discussion. Being explicit in this regard is important because we will be performing analyses and making distinctions which are uncommon and sensitive to differences which are usually immaterial.

1.4.1 Continuation semantics

Programs contain not only code which manipulates data, but also code which manipulates control-flow, that is, determines which code will be executed. The latter sort of code is formed by *control constructs*: conditional branches or procedures, for instance. When giving the semantics of data manipulation code, some auxiliary information is needed: the *data context*. Similarly, when giving the semantics of control-flow manipulation code, a *control context* is needed. For instance, knowledge of the branches of a conditional is needed to determine what code should be executed after evaluating the test, and the return address of a procedure is needed to determine what code should be executed once the procedure returns.

Commonly, a data context consists of a (data) *environment* and a (data) *state*, and similarly, a control context consists of a *control environment* and a *control state*. The control environment is used to handle constituents of the control context which exhibit a binding-like behavior. For instance, in a language with the ability to jump to labeled commands, determining which command a label refers to is treated similarly to standard lexical binding of identifiers. Also, determining the return address of a procedure call is handled much like dynamic identifier binding. On the other hand, the control state is used for constituents whose behavior is less regular or predictable. For example, in a language with coroutines, keeping track of how far each coroutine has executed is handled in a stateful manner.

It is not always be immediately clear what the control context should contain. One helpful, we find, way to think about it is that the control context contains those resources which govern or enable control flow. But this is admittedly unacceptably vague, and we will return to this point in Chapter 12.

Continuations are the raw material of control. Hence, control contexts are generally one or more continuations, combined in various ways, depending on the language and sort of code under consideration. A *continuation* consists of a piece of code together with the portion of the code's (data and control) context which is known when the continuation is constructed: (part of) the data environment, and (part of) the control environment; and is parameterized by the portion of the code's context which is unknown when the continuation is constructed: (part of) the data environment, (part of) the control environment, and the data and control states.[4]

Once constructed, the only operation on a continuation is to *invoke* it by supplying arguments and then executing its code. So intuitively, a continuation is the destination of a "jump with arguments." Crucially, after arguments have been passed to a continuation, the entire context (data and control) of the continuation's code is at hand, and so there is enough information to execute all the way to "the end." Hence, invoking a continuation causes the remaining computation to be performed in its entirety.

An effectively equivalent but specific and highly syntactic characterization of continuations as "evaluation contexts" is developed in [Fel87]. A more implementation-oriented view of continuations can also be taken, but there is a potential point of confusion.[5]

[4] Depending on the setting, there is some freedom to choose whether a piece of context is kept internal to, or passed as an argument to, a continuation. For instance, delaying the construction of a continuation may allow a piece of context to be kept internally rather than be passed as an argument. An example of this appears in Chapter 3, and this flexibility is used in an entirely different fashion in Chapter 5.

[5] In continuation-based compilation [App92, KKR+86, Ste78], a continuation is commonly repre-

The self-contained nature of continuations, and their extremely narrow interface also allows a very abstract view to be taken. Type-theoretically, we can define a continuation type constructor which, for any type T, forms the type

$$\neg T$$

of T-accepting continuations. We could then provide terms to construct and invoke continuations of these types. This line of development is taken in [DHM91, Thi97] and while we view this as the conceptually correct abstract path, in order to facilitate more direct reuse of existing linear logic and typing work, we use a functional representation of the type of continuations:

$$\neg T \stackrel{\text{def}}{=} T \to \mathbf{R}$$

Here, the parameter-passing aspect of continuations is modeled by the ordinary, intuitionistic, function type $(\cdot) \to (\cdot)$; and so ordinary λ-abstraction and -application can be used to construct and invoke continuations. Treating the type of results $\mathbf{R}$ for the return type of continuations as if abstract (discussed below), is intended to capture how, once invoked, a continuation runs all the way to "the end," that is, invoking a continuation does not return an intermediate result but instead produces the final result, or answer, of the entire computation.

Results, that is, elements of $\mathbf{R}$, represent the effect of executing a program as viewed from outside the program. So for a programming language with an output facility, the result type could be represented as a (possibly infinite) sequence of the output values (possibly terminated with an indicator of program termination) and would have an operation to add a value to a sequence. The result type should be abstract since programs are not directly privy to their surroundings' view of their actions, among other reasons. If no operations on the result type are needed, then a free type identifier, or a type with no elements, can be used. Intuitively, the sense of abstraction we mean here is that the result type is treated uniformly: a computation cannot branch on, or even freely produce, values of result type.

Note that since we treat the result type abstractly, we do not want to form closed terms of type $\mathbf{R}$, and so a program must be provided with a means of producing a result. Almost universally, this is accomplished by parameterizing programs with a *toplevel* (or *initial*) continuation, which the program invokes to terminate. This continuation contains operating system, say, code in which $\mathbf{R}$ is concrete. So in the output example above, no $\mathbf{R}$-operation to indicate program termination is needed, since when a program terminates it will invoke its toplevel continuation, which can manipulate the representation of $\mathbf{R}$ directly. In other words, the toplevel continuation *is* the $\mathbf{R}$-operation for termination.

A *continuation semantics* of a programming language is a semantics where the meaning of each piece of code maps its control context to the meaning of the whole computation. Hence, the control context is an abstraction of the effect of executing "the rest of the computation." For many languages, the control context can be uniformly represented as a continuation, so the two are often conflated and "continuation semantics" is named as it is. Some authors also prefer to define the term "continuation" to mean what we have termed control context. However, we will consider some languages where the control context is represented by several continuations, and so it is necessary to distinguish between continuations and control contexts in general. In

sented as a particular sort of closure, and invoking a continuation is effected by setting up the arguments in the machine registers and then jumping to the code referenced by the closure. An absolutely crucial point, however, is that continuations are viewed "deeply," as in the difference between deep and shallow copying of linked data structures. This means that if an identifier references a continuation closure which references another continuation closure (as part of the control environment, for instance), and so on, then the continuation denoted by the identifier is (represented by) the entire list of continuation closures, not just the first one. This may seem pedantic, but the linear type system we will present will be used to ensure properties such as "continuations are not copied," and in doing so it takes the deep view. To facilitate handling pieces of control context shallowly, we use delimited continuations, see Chapter 5.

particular, we will consider many cases in which the continuations do not represent an abstraction of the effect of executing "the rest of the computation."

Often, a continuation semantics takes the form of a *transformation* (or translation or compilation) from the programming language under study, called the *source* language, to an appropriate semantic metalanguage (or intermediate language) with well-understood meaning (or implementation), called the *target* language. Such a *continuation-passing style* (CPS) transformation makes all transfers of control explicit and uniform using continuations. The range of a CPS transformation, as a language, is sometimes referred to simply as CPS.

1.4.2 Refining with linearity

The usual typed λ-calculus is the standard choice for the target language of a CPS transformation. The range of the transformation will inhabit certain target language types, and a *refined* semantics can be given by using more restrictive, but still sufficiently expressive, versions of the exercised target types. For a typed source language, the exercised types will be CPS versions of the source language types, while untyped source languages are treated as if universally typed. So, even if the control behavior of the source language (the patterns of control-flow exhibited by source programs) is independent of typing issues, analysis of the control behavior can use types in the target language.

Aiming to define restricted target types, we use a λ-calculus target language which includes both ordinary (intuitionistic) and restricted (linear or affine) function types. A *linear* function, very roughly, can neither duplicate nor discard its parameter, while an *affine* function, very roughly, cannot duplicate, but can discard, its parameter [Gir87]. Unfortunately, there is some divergence in the literature on the terminology used to describe where linearity resides. Some authors refer to the parameter of a function of type $A \multimap B$ as being linear (respectively, affine), rather than the function itself. We are following Girard's original usage: a function of type $A \multimap B$ is a linear (respectively, affine) function, and so the parameter is *used* linearly (respectively, affinely). In the context of semantics or typing, we often use the terms *refined* or *restricted* to mean either linear or affine.

This choice of target language enables us to restrict to linear/affine use of control contexts by strengthening key intuitionistic types of the standard semantics to linear/affine analogues. The chief effect of this restriction is the resulting inability to invoke a continuation multiple times, an ability which is crucial to the expressive power of unconstrained continuations [RT99, Thi99b, Thi00]. As mentioned above, this refinement of types, restricting the use of control contexts, is an exploitation of the stylized use of continuations that many control behaviors exhibit.

1.5 Motivations

While we do not intend to offer a thorough motivation of constraining control behavior by linearly typing CPS, let alone justifying why one might care about CPS or continuation semantics to begin with, we will detail some surface scratches.

From the beginning, it has been recognized that CPS is more expressive in terms of control behavior than most source languages. From this recognition there has been a thread in the literature presenting ideas about continuation usage, usually under the tag-line "one continuation [identifier] is enough" [CHO99, Dan92, Dan94, Dan00, DDP00, DL92, DP95, FSDF93, SF93]. Formalization of such ideas about continuation usage, some old and some new, with restricted type systems is a central aspect of this work. This focus on static typing sets our approach apart from earlier work on constraining the power of continuations [BWD96, FH85], where the constraints generally take the form of assertions, checked at runtime, which ensure a program's dynamic behavior obeys certain invariants. Relying on a static type system yields many of the usual advantages (static checkability, unnecessity of runtime checks, etc.) and disadvantages

(loss of expressiveness, etc.) of static typing. Also, since "used linearly" and "invoked exactly once" are not the same, as discussed in Section 3.4, the results from prior work in this vein do not immediately carry over.

But why formalize? And why with type systems? There may be an argument that formalization is self-motivating, but we do not want to appeal to it.

1.5.1 Conceptual subtleties

The first reason is a traditional one: formalization can help us identify and understand the conceptual subtleties. In particular, if control constructs use continuations in a stylized, restricted, way, then we may hope to better understand these constructs by studying the typing properties of their semantics. An example of this is contained in the observation that first-class continuations break linear typing, while exceptions do not. Another example is the importance of the distinction between the notions of "control context" and "continuation" we will argue for in Chapter 5. And, as in Thielecke's [Thi02], we will also see that the semantics of binding is immensely important to the semantics of control; in our case, to the admissibility of refined interpretations.

1.5.2 Mechanized reasoning

Recently a more pragmatic motivation for formalization has arisen, and one which motivates formalizing ideas with type systems. Increasing degrees of reasoning about programs are being mechanized. For instance, witness the trend: C [KR88], ML [MTHM97], Proof Carrying Code [Nec97], and beyond [App01]. In order to support this, the properties to be reasoned about must be expressed in a language which tools can handle, instead of the informal and highly undecidable language of mathematical discourse. For type preserving or certifying compilers (for instance [MWCG99, NL98, TMC+96]); or other automated (not necessarily automatic) checkers, analyzers, or verifiers; knowing a property holds, even with a proof, is not enough: the property must be captured by the tool's formalism. If an inexpressible property is taken advantage of, then at some point the tool will require reasoning outside its formal system, causing problems. Hence more formality is needed.

A chief goal of this study of restricting CPS with linear or affine type systems is to provide some background and foundation for this sort of formalization of useful properties. One instance we present is a precise characterization of the range of the CPS transformation using an affine type system. Here, by precise we mean that all terms in the target language come from some term in the source language. This provides a sort of completeness, or fullness, result (no junk) which, while not the most theoretically desirable due to a heavy restriction on types and its syntactic nature, ought to be good enough to reduce loss of precision in compiler analyses. But the point is that the junk-free target language is defined with a type system; that is, in a way quite accessible to tools.

As an example of a property this formalization may allow to be exploited, in a certifying compiler which wishes to stack-allocate activation records, the type systems of the intermediate languages must be tight enough to exclude all programs which violate this discipline. Unlike in a standard compiler, it is not sufficient to simply "know" that none of the intermediate language programs which actually come from earlier stages of the compiler will exhibit such behavior: in a certifying compiler the stage which makes this assumption will generate code which fails to typecheck, and hence no proof of the generated code can be produced.

1.5.3 Loss of precision

In a way, this formalization for mechanized reasoning is an instance of the general problem of loss of precision when translating to CPS. For instance, consider a compiler or other program analysis or verification system. Here CPS can be very useful since it provides a uniform mechanism for all control flow and some language features such as higher-order procedures become much

more manageable. This simplification comes at a price in precision, however, since the standard, unrestricted, CPS is usually (much) more expressive than the fragment needed to interpret the source language in question. Often this loss of precision is unacceptable. Use of a linear or affine type system for CPS restricts the target language, reducing its expressiveness and, in turn, the loss of precision.

In fact, in recent work, Zdancewic and Myers use a nearly identical system to prove secure information flow in a higher-order, imperative language [ZM02]. The constrained use of continuations is crucial to their proof, providing an instance where linear continuation usage recovers enough precision.

1.6 Contributions

The path we take in exploring restricted use of control contexts is driven primarily by the character of the semantics involved, rather than by established language features or constructs. We consider various control constructs in order to have the simplest possible languages with which to think about the semantics. A result of this is that some of the languages we consider are rather odd, but we end with familiar—if drastically simplified—control constructs. It also cannot be denied that our choices of source languages have been influenced by our motivations in the realm of mechanized reasoning for Proof Carrying Code, Typed Assembly Language, etc.

As a result of our focus on form of semantics over form of language, we view the primary contribution of this work to be not the restricted treatments of particular language constructs or features, but a collection of ideas on how to treat semantics of various forms. We have aimed for sufficiently broad coverage of semantic forms to allow others to treat the control constructs of their interest by analogy with a language with similar semantic form.

We now briefly summarize in anticipation by giving the type (or domain) which drives each interpretation, and highlight the concepts and pitfalls associated with each.

procedures, simply-typed

$$\overline{S \to T} \stackrel{\text{def}}{=} \underbrace{(\overline{T} \to \mathbf{R})}_{\substack{\text{return}\\ \text{continuation}}} \multimap \underbrace{\overline{S} \to \mathbf{R}}_{\substack{\text{call}\\ \text{continuation}}}$$

procedures, untyped

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to \mathbf{R})}_{\substack{\text{return}\\ \text{continuation}}} \multimap \underbrace{D \to \mathbf{R}}_{\substack{\text{call}\\ \text{continuation}}}$$

The semantics for the simply-typed and untyped incarnations of procedures are very similar: in both cases the key is a type of continuation transformers which forces the return continuation to be used linearly. This restriction exactly captures the stylized use inherent in the procedure call and return mechanism: Once a procedure returns to a call site, that site will not be returned to again; and once a procedure is called, it will return (or diverge).

Recursion in the untyped case introduces two potential pitfalls: Do recursive calls result in a continuation being invoked more than once? And: Does divergence lead to ignoring a continuation, requiring affine typing instead of linear? In short: No. This is our first encounter with the fact that, for continuations, "used linearly" and "invoked exactly once" have wildly different meanings.

In both cases, we prove soundness. The transformation is standard in both cases and so computational adequacy is not a question. We consider completeness for the untyped case, which generalizes to the simply-typed case.

exceptions

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to \mathbf{R})}_{\substack{\text{return}\\ \text{continuation}}}\ \&\ \underbrace{(D \to \mathbf{R})}_{\substack{\text{handler}\\ \text{continuation}}}\ \multimap\ \underbrace{D \to \mathbf{R}}_{\substack{\text{call}\\ \text{continuation}}}$$

This semantics is the first where the notions of control context and continuation do not coincide. Here the control context is an additive pair of the return and handler continuations rather than just the return continuation. Again, the restricted typing exactly captures the stylization inherent in the source language: When a procedure is called, it may either return normally or raise an exception, but not both.

As with procedures, we consider soundness and completeness, while adequacy would be a slight extension of existing work.

procedures, delimited continuation interpretation

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to \underbrace{(D \to \mathbf{R})}_{\text{toplevel continuation}} \multimap \mathbf{R})}_{\substack{\text{return}\\ \text{delimited continuation}}} \multimap\ \underbrace{D \to \underbrace{(D \to \mathbf{R})}_{\text{toplevel continuation}} \multimap \mathbf{R}}_{\substack{\text{call}\\ \text{delimited continuation}}}$$

This semantics makes the implicit dependence of a continuation on the toplevel continuation explicit in the types. This is accomplished by using delimited continuations, which take a toplevel continuation as an argument rather than depend on it through the environment. This distinction is not particularly revealing in the case of procedures, but we rely on it crucially in later interpretations.

first-class continuations

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to (D \to \mathbf{R}) \multimap \mathbf{R})}_{\substack{\text{return}\\ \text{delimited continuation}}} \to\ \underbrace{D \to (D \to \mathbf{R}) \multimap \mathbf{R}}_{\substack{\text{call}\\ \text{delimited continuation}}}$$

Using this type, we interpret `call/cc` and `abort` while using continuations linearly, though use of delimited continuations is unrestricted. This semantics serves to drive home the importance of the distinction between control contexts and continuations. Since the return delimited continuation is part of the control context, and its use is unrestricted, we can interpret full first-class control. So there is a potential pitfall where, unless the entire control context is correctly identified and restricted, linear typing need not be a restriction at all.

simple command language

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{current}\\ \text{continuation}}}\ \multimap \mathbf{S} \to \mathbf{R}$$

In preparation for treating more interesting command languages, we show how a simple command language with essentially no control structure admits a linear interpretation using this semantics. In this case we prove soundness and adequacy.

forward jumps

$$\underbrace{\&\,(\mathbf{S} \to \mathbf{R})}_{\substack{\text{labeled}\\ \text{continuations}}} \;\&\; \underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{current}\\ \text{continuation}}} \;\multimap \mathbf{S} \to \mathbf{R}$$

Technically, this semantics is just a mild extension of that used for exceptions, which roughly corresponds to having multiple sorts of exceptions. It may not be apparent from the type, but the primary point of this semantics is to treat nested scopes binding continuations which are passed downward only. One point to note is that the additive tuple of labeled continuations is a sort of environment, and would generally be made implicit. But to give a linear interpretation the labeled continuations and current continuation must be treated symmetrically, forcing an explicit environment semantics. As usual, we prove soundness and adequacy.

backward jumps

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{labeled}\\ \text{continuation}}} \;\&\; \underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{current}\\ \text{continuation}}} \;\multimap \mathbf{S} \to \mathbf{R}$$

While this type is not very interesting, it suffices to allow investigation of explicit recursion rather than the recursion implicit in the self-application of untyped procedures. The semantic difference is that the former naturally leads to recursively defined continuations, while the latter involves only recursive continuation transformers. But recursive continuations break linear typing, so we must partially specify the result type and instead define recursive delimited continuations (which here are just continuation transformers). But care must be taken to choose the control context correctly, which leads to taking a fixed-point of a term of type

$$((\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}) \multimap (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}$$

Here it is not immediately obvious that the principal $\multimap$ must not be an $\to$. Additionally, taking fixed-points at this type requires a fixed-point combinator with nonstandard type. Finally, this semantics requires an affine, rather than linear, type system. We prove soundness and adequacy.

coroutines

$$\underbrace{K}_{\substack{\text{running}\\ \text{delimited}\\ \text{continuation}}} \;\multimap K$$

where

$$K \stackrel{\text{def}}{=} \mu K.\, \mathbf{S} \to \underbrace{K}_{\substack{\text{blocked}\\ \text{delimited}\\ \text{continuation}}} \;\multimap \underbrace{(\mathbf{S} \to K \multimap \mathbf{R})}_{\text{shared continuation}} \;\multimap \mathbf{R}$$

This semantics is the first in which (delimited) continuations are upward, that is, arguments to other (delimited) continuations. For this reason, K must be recursive. In this semantics the upward (delimited) continuations play the role of a control store, interpreting (two) coroutines. The use of delimited continuations here is crucial. Since there are two coroutines, each with its control state represented by a continuation, if undelimited continuations

were used, then the toplevel continuation would have to be shared between them, which linear typing is not happy with. Instead we use delimited continuations which allow us to separate out the toplevel continuation and pass it back and forth between the coroutines as they are active, avoiding the need to actually share it. This semantics is affine since the control state of one coroutine must be discarded when the other terminates. We prove soundness and adequacy.

stored global labels, and dynamically-assigned labels

$$\underbrace{(\mathbf{N} \to K)}_{\substack{\text{label}\\ \text{environment}}} \;\&\; \underbrace{K}_{\substack{\text{current}\\ \text{continuation}}} \;\multimap K$$

We use this semantics for two interpretations. The main issue in the first is how to treat computed jumps. In previous semantics, where the control environment was represented with an additive tuple, computed jumps could not be expressed since projections from such tuples rely on compile-time substitution. So instead we use an intuitionistic function type with equivalent typing restrictions.

In the second, the goal is to come as close to providing a control store with a semantics which manipulates a control environment. We do this by considering a language with nested blocks of dynamically-assigned (or fluidly-bound) labels, which is very similar to a stripped-down multiple-exceptions mechanism.

stored commands

$$\underbrace{K}_{\substack{\text{current}\\ \text{continuation}}} \;\multimap K$$

where

$$K \stackrel{\text{def}}{=} \mu K.\; \underbrace{\mathbf{S}}_{\substack{\text{data}\\ \text{store}}} \to \underbrace{(\mathbf{N} \to K \multimap K)}_{\substack{\text{code}\\ \text{store}}} \to \mathbf{R}$$

This semantics has a store component involving continuations, but the key is that it is commands, that is, continuation transformers, rather than continuations, which are stored. The significance of the distinction is that commands are just inert data, while continuations encode control points, and hence a code store is not part of the control context and need not be used in a restricted fashion.

mutable labels

$$\underbrace{K}_{\substack{\text{current}\\ \text{continuation}}} \;\multimap K$$

where

$$K \stackrel{\text{def}}{=} \mu K.\; \underbrace{\mathbf{S}}_{\substack{\text{data}\\ \text{store}}} \to \underbrace{\textstyle\bigotimes_n K}_{\substack{\text{control}\\ \text{store}}} \multimap \mathbf{R}$$

Finally, we consider a semantics similar to that used for coroutines but with a more general control store. The point of considering this semantics is primarily to illustrate the difficulty a source language has of meeting its constraints. As a result, while the source language of mutable labels can express coroutines, it is unsafe and all the operations on the control store must move, and never copy, the stored labels.

Chapter 2

Procedures, Simply-Typed

In this chapter we expand the illustration of the central idea of this work presented in Chapter 1 by fleshing out the treatment of simply-typed λ-calculus and arguing that it is sound (typechecks). We present the basic details and intuitions of the linear type system of the target language, and of the CPS transformation into it. The conceptual core of the chapter, however, is the explanation of how linearity corresponds to the stylized usage of continuations implicit in the procedure call and return mechanism.

Chapter 3 generalizes this treatment to untyped λ-calculus, however, since doing so requires the additional complication of recursion and recursive types, we first introduce the ideas in the simply-typed setting. Also, explicitly describing the correspondence between types in the source and target languages may be helpful to some readers.

2.1 Source Language

The source language is simply-typed, left-to-right, call-by-value λ-calculus. This language is entirely standard and so we only briefly summarize the formulation of its syntax and type system we use. (Additionally, various uninteresting λ-calculus technicalities of this language carry over from the definition of the target language given in Appendix A.)

2.1.1 Syntax

The syntax of terms is given by the grammar

$$\begin{array}{lll} M ::= & & \textit{terms} \\ & \mid x & \text{identifier} \\ & \mid \lambda x.\, M & \text{abstraction} \\ & \mid M\, M & \text{application} \end{array}$$

Convention: The body of an abstraction extends as far to the right as possible, so $\lambda x.\, M\, N$ parses to $\lambda x.\, (M\, N)$ rather than $(\lambda x.\, M)\, N$. Application is left-associative, so $M\, N\, O$ parses to $(M\, N)\, O$ rather than $M\, (N\, O)$.

As an example, consider the term:[1]

$$\big(\lambda double.\, double\, (\lambda y.\, y - 12)\, 66\big)\, \big(\lambda f.\, \lambda x.\, f\, (f\, x)\big) \tag{2.1}$$

[1] We use an integer base type in the example, despite not treating it formally.

which first binds $double$ to the doubling function and then calls it with the decrement-by-12 function and 66. Once $double$ is called, the decrement-by-12 function is called with 66. Then 54 is returned to the inner call site, at which point the decrement-by-12 function is called again, but now with 54. Finally, 42 is returned to the outer call site, and again returned to the original site from which $double$ was called.

2.1.2 Type system

The syntax of types is given by the grammar

$T ::=$		*types*
	$\mid \mathbf{N}$	base type
	$\mid T \to T$	procedure type

Convention: The procedure type constructor is right-associative, so $S \to T \to U$ parses to $S \to (T \to U)$ rather than $(S \to T) \to U$.

At this point we include a base type simply to keep the type system from collapsing to a degenerate case, and so do not provide any base values or operations.

The judgment $\Gamma \vdash M : T$ states that M is a well-typed term of type T in *context* Γ (which is a finite set of typings such that no identifier is the subject of more than one typing)[2]:

$$[\textsc{Id}]\ \frac{}{\Gamma, x : T \vdash x : T} \qquad [\textsc{Abs}]\ \frac{\Gamma, x : S \vdash M : T}{\Gamma \vdash \lambda x.\, M : S \to T} \qquad [\textsc{App}]\ \frac{\Gamma \vdash M : S \to T \quad \Gamma \vdash N : S}{\Gamma \vdash M\, N : T}$$

2.2 Target Language $(= \mathbf{R} + (\cdot) \to (\cdot) + (\cdot) \multimap (\cdot))$

The target language is a formulation of linear type theory based on DILL [BP97], which is a presentation of linear typing that allows a pleasantly direct description of $(\cdot) \to (\cdot)$ (which does not rely on decomposition through $!(\cdot)$). To interpret simply-typed λ-calculus we do not need product types, recursive types, etc., and so defer their introduction until they are needed. (We present the entire target language, and various technicalities, in Appendix A.)

2.2.1 Syntax

The syntax of terms is given by the grammar

$M ::=$		*terms*
	$\mid x$	identifier
	$\mid \lambda x.\, M$	(ordinary, intuitionistic) abstraction
	$\mid \delta x.\, M$	restricted (linear) abstraction
	$\mid M\, M$	(ordinary, intuitionistic) application
	$\mid M \llcorner M$	restricted (linear) application

Convention: Restricted abstraction and restricted application follow conventions similar to abstraction and application.

[2] See Section A.3 for details on contexts, typings, and subjects.

In Chapter 3 we will present this language's equational theory, but for now an informal description of the intended semantics suffices. Ordinary abstraction and application is the usual, intuitionistic, call-by-name one. Linear abstraction and application is operationally equivalent to the intuitionistic version, the difference is in typing only. We use δ to form linear abstractions since the linear abstractions we shall use are commonly continuation transformers, which are effectively the difference, or *delta*, between two continuations.[3]

2.2.2 Type system

The syntax of types is given by the grammar

$A, P ::=$		*types*
	$\mid \mathbf{N}$	base type
	$\mid \mathbf{R}$	result (answer) type
	$\mid A \to A$	(ordinary, intuitionistic) function type
	$\mid P \multimap P$	restricted (linear) function type

Convention: The restricted function type constructor follows the same conventions, and has the same precedence, as the ordinary function type constructor.

The reason we use two type metaidentifiers, A and P, will be explained in Chapter 3. The distinction has to do with recursion and is irrelevant for the moment since we have not yet introduced recursive types.

The judgment $\Gamma\ ;\ \Delta \vdash M : A$ states that M is a well-typed term of type A in context $\Gamma\ ;\ \Delta$:

$$[\textsc{Id}]\ \frac{}{\Gamma\ ,\ x : A\ ;\ - \vdash x : A} \qquad [\textsc{RId}]\ \frac{}{\Gamma\ ;\ x : P \vdash x : P}$$

$$[\textsc{Abs}]\ \frac{\Gamma\ ,\ x : A\ ;\ \Delta \vdash M : B}{\Gamma\ ;\ \Delta \vdash \lambda x.\, M : A \to B} \qquad [\textsc{App}]\ \frac{\Gamma\ ;\ \Delta \vdash M : A \to B \quad \Gamma\ ;\ - \vdash N : A}{\Gamma\ ;\ \Delta \vdash M\, N : B}$$

$$[\textsc{RAbs}]\ \frac{\Gamma\ ;\ \Delta\ ,\ x : P \vdash M : Q}{\Gamma\ ;\ \Delta \vdash \delta x.\, M : P \multimap Q} \qquad [\textsc{RApp}]\ \frac{\Gamma\ ;\ \Delta \vdash M : P \multimap Q \quad \Gamma\ ;\ \Delta' \vdash N : P}{\Gamma\ ;\ \Delta\ ,\ \Delta' \vdash M \_ N : Q}$$

The abstraction rules, [Abs] and [RAbs], show that typings are added to whichever zone is appropriate to the abstraction: intuitionistic zone for intuitionistic abstraction and linear zone for linear abstraction. The most significant restriction of the system is made by [App]: the argument to an intuitionistic function cannot depend on any controlled resources (identifiers in the linear zone). Since an intuitionistic function's use of its parameter is unconstrained, the function may duplicate (or discard) its parameter; and if the argument were to contain a controlled resource, then duplicating (discarding) the parameter would result in duplicating (discarding) the controlled resource as well. In [RApp], the restriction that the operator and operand depend on disjoint controlled resources is also crucial to prohibiting duplication. Finally, [Id] requires the linear zone to be empty and [RId] requires the linear zone to be a singleton; which is essential to prevent discarding of controlled resources.

Taken together, these restrictions preclude Contraction and Weakening in the linear zone, meaning that

[3] Although the technical definition differs, conceptually this notion of difference is similar to that in [MQ94], so this slight overloading of the term "difference" seems justified.

$$[\text{RCONT}]\ \frac{\Gamma\ ;\ \Delta\ ,\ x:A\ ,\ y:A \vdash M:B}{\Gamma\ ;\ \Delta\ ,\ x:A \vdash M[y \mapsto x]:B} \qquad [\text{RWEAK}]\ \frac{\Gamma\ ;\ \Delta \vdash M:B}{\Gamma\ ;\ \Delta\ ,\ x:A \vdash M:B}$$

are *not* admissible rules. On the other hand, Contraction and Weakening in the intuitionistic zone

$$[\text{CONT}]\ \frac{\Gamma\ ,\ x:A\ ,\ y:A\ ;\ \Delta \vdash M:B}{\Gamma\ ,\ x:A\ ;\ \Delta \vdash M[y \mapsto x]:B} \qquad [\text{WEAK}]\ \frac{\Gamma\ ;\ \Delta \vdash M:B}{\Gamma\ ,\ x:A\ ;\ \Delta \vdash M:B}$$

are built into the system and we will often implicitly contract or weaken intuitionistic zones. Also, note that since contexts are built from sets (which are unordered), the Exchange rules

$$[\text{EXCH}]\ \frac{\Gamma\ ,\ y:B\ ,\ x:A\ ,\ \Gamma'\ ;\ \Delta \vdash M:C}{\Gamma\ ,\ x:A\ ,\ y:B\ ,\ \Gamma'\ ;\ \Delta \vdash M:C} \qquad [\text{REXCH}]\ \frac{\Gamma\ ;\ \Delta\ ,\ y:B\ ,\ x:A\ ,\ \Delta' \vdash M:C}{\Gamma\ ;\ \Delta\ ,\ x:A\ ,\ y:B\ ,\ \Delta' \vdash M:C}$$

are built into the system.

2.3 Refined CPS Transformation

The CPS transformation is Fischer's [Fis93] continuation-first transformation cast in terms of the linear target language. As expected, the interpretation revolves around the treatment of procedures. In this case, the control context of each source term is a continuation which represents the computation which will occur once the term's value has been computed. A procedure may be called from many different sites in the code (for instance, the two call sites of f in (2.1)), and so its body may be executed in the context of different continuations, potentially one for each call site. This is significant since it means that a procedure's *return* continuation, the continuation which should be invoked once the procedure's body has been executed, is unknown at CPS transformation-time.[4] Hence, the interpretation of a procedure is parameterized by its return continuation.

From a *direct style* (DS), as opposed to CPS, point of view, the control context of a procedure call is commonly represented as a call stack referenced by a return address.[5] Implicit in the procedure call mechanism is the fact that knowledge of the return address is transmitted from the caller to the callee through the store. That is, in DS the control context is passed statefully, and so CPS is simply making this information-flow visible at the level of types.

More formally, the transformation of types [MW85], shown in Figure 2.1, indicates that the base types in the source and target languages are essentially identical

$$\overline{\mathbf{N}} \stackrel{\text{def}}{=} \mathbf{N}$$

and that source procedures are interpreted by *continuation transformers*[6]

$$\overline{S \to T} \stackrel{\text{def}}{=} \underbrace{(\overline{T} \to \mathbf{R})}_{\substack{\text{return}\\ \text{continuation}}} \multimap \underbrace{\overline{S} \to \mathbf{R}}_{\substack{\text{call}\\ \text{continuation}}}$$

[4] Duplicating code would allow this continuation to be determined, but this technique is unrealistic and does not work in the presence of recursion.

[5] Taking the implementation-oriented view of a continuation as a list of continuation closures, if this list is stack-allocated, then the traditional call stack reemerges [Dan00].

[6] Several types isomorphic to this are also standard and discussed in Section 3.7.

which map continuations to continuations. The CPS version of a procedure first accepts a return continuation. After the return continuation, the argument proper is accepted and execution proceeds. So we refer to the continuation resulting from parameterizing a procedure with its return continuation as the *call* continuation. When the value to be returned has been computed, the return continuation is invoked with it, at which point the computation represented by the return continuation will occur. So CPS decomposes the procedure call mechanism into two jumps with arguments: one from the caller to the callee for the call, passing the argument; and one from the callee to the caller for the return, passing the return value. Further discussion of this decomposition can be found in [Thi99a].

The CPS transformation proper maps a source judgment

$$\Gamma \vdash M : T$$

to a target judgment

$$\overline{\Gamma} \; ; - \vdash \overline{M} : (\overline{T} \to \mathbf{R}) \multimap \mathbf{R}$$

using the transformation of types discussed above, the transformation of contexts given in Figure 2.1 which simply transforms all the types in the context, and the transformation of terms discussed below.

As indicated above, terms of type T are interpreted with the type $(\overline{T} \to \mathbf{R}) \multimap \mathbf{R}$, essentially treating them as a sort of thunk: T-returning parameterless procedures. That is, if a unit type $\mathbf{1}$ was added to the source language, the type of T-returning parameterless procedures would be $\mathbf{1} \to T$, and the CPS version of this would then be $(\overline{T} \to \mathbf{R}) \multimap \mathbf{1} \to \mathbf{R}$, which is isomorphic to $(\overline{T} \to \mathbf{R}) \multimap \mathbf{R}$. This interpretation is somewhat roundabout and leads to many "administrative" redexes [Plo75]; and there exist more direct and refined "compacting" transformations [DF92, HL93, SF93]. However, the other transformations are more complicated, and furthermore, conducting our analysis on such a transformation would likely necessitate a multilevel type system in the target language. Issues of administrative redexes are orthogonal to our analysis and the use of a multilevel type system for CPS has been investigated elsewhere [PP00, PY01], so we prefer simple, if somewhat indirect, transformations.

Concretely, the transformation of terms is shown in Figure 2.1. The CPS transformation of an identifier

$$\overline{x} \stackrel{\text{def}}{=} \delta k.\, k\, x$$

being a value, accepts the return continuation and immediately passes it the CPS version of the identifier, which is simply the identifier itself since we have a substitution-based call-by-value semantics. That is, since the semantics is substitution-based, identifiers are used only to direct substitution, and since the semantics of the source is call-by-value, only CPS values will be substituted for (intuitionistic) identifiers.

The CPS transformation of an abstraction

$$\overline{\lambda x.\, M} \stackrel{\text{def}}{=} \delta k.\, k\, \delta h.\, \lambda x.\, \overline{M} \llcorner h$$

also a value, accepts the return continuation and immediately passes it the CPS version of the abstraction. The immediacy of this invocation corresponds to the fact that evaluating a value is trivial. This invocation effects the return jump in the decomposition of the procedure call mechanism. The CPS version of an abstraction first accepts the return continuation h, yielding another continuation. This is the call continuation, which effects the call jump in the decomposition of the procedure call mechanism when invoked with the argument x. The body M is then executed with the return continuation h: $\overline{M} \llcorner h$.

Finally, the transformation of an application

$$\overline{M\, N} \stackrel{\text{def}}{=} \delta k.\, \overline{M} \llcorner \lambda m.\, \overline{N} \llcorner \lambda n.\, m \llcorner k\, n \quad m \notin \mathsf{fi}(N)$$

Figure 2.1 Refined CPS of Simply-Typed λ-calculus

Types

$$\overline{\mathbf{N}} \stackrel{\text{def}}{=} \mathbf{N}$$
$$\overline{S \to T} \stackrel{\text{def}}{=} \underbrace{(\overline{T} \to \mathbf{R})}_{\text{return}} \multimap \underbrace{\overline{S} \to \mathbf{R}}_{\text{call}}$$

Contexts

$$\overline{x_1 : T_1 \, , \ldots , x_n : T_n} \stackrel{\text{def}}{=} x_1 : \overline{T_1} \, , \ldots , x_n : \overline{T_n} \quad (n \geq 0)$$

Terms

$$\Gamma \vdash M : T \quad \text{transforms to} \quad \overline{\Gamma} \; ; - \vdash \overline{M} : (\overline{T} \to \mathbf{R}) \multimap \mathbf{R}$$

$$\begin{aligned} \overline{x} &\stackrel{\text{def}}{=} \delta k.\, k\, x \\ \overline{\lambda x.\, M} &\stackrel{\text{def}}{=} \delta k.\, k\, \delta h.\, \lambda x.\, \overline{M} \llcorner h \\ \overline{M\, N} &\stackrel{\text{def}}{=} \delta k.\, \overline{M} \llcorner \lambda m.\, \overline{N} \llcorner \lambda n.\, m \llcorner k\, n \quad m \notin \mathsf{fi}(N) \end{aligned}$$

Programs

$$- \vdash M : \mathbf{N} \quad \text{transforms to} \quad - \; ; - \vdash \lfloor M \rfloor : (\mathbf{N} \to \mathbf{R}) \multimap \mathbf{R}$$

$$\lfloor M \rfloor \stackrel{\text{def}}{=} \overline{M}$$

accepts the return continuation k and then executes M with continuation $\lambda m.\, \overline{N} \llcorner \lambda n.\, m \llcorner k\, n$, which, when invoked with the value m computed by M, executes N with continuation $\lambda n.\, m \llcorner k\, n$, which, when invoked with the value n computed by N, applies the operator m to its return continuation k, yielding the call continuation $m \llcorner k$ which is then invoked with the argument n.

Once the CPS transformation has been defined, the semantics of closed terms of base type, that is, programs, is trivial (although we have not included any syntax to form them). As usual, we interpret source programs by results parameterized by a toplevel continuation:

$$\underbrace{(\mathbf{N} \to \mathbf{R})}_{\substack{\text{toplevel} \\ \text{continuation}}} \multimap \mathbf{R}$$

The semantics is shown in Figure 2.1. The reason for stating this semantics is that it makes clear that programs are not self-contained but instead depend on the operating system, say, to provide them with a way to terminate by jumping back into the operating system with an argument.

2.4 Soundness

We now argue that the interpretation presented in the previous section is sound, meaning that the transformation obeys the stated typing constraints.

Proposition 1 (Soundness) *1. For any source term M, if $\Gamma \vdash M : T$, then*

$$\overline{\Gamma} \; ; - \vdash \overline{M} : (\overline{T} \to \mathbf{R}) \multimap \mathbf{R}$$

2. *For any source program M, if $- \vdash M : \mathbf{N}$, then*

$$- \,;\, - \vdash \lfloor M \rfloor : (\mathbf{N} \to \mathbf{R}) \multimap \mathbf{R}$$

The key to the soundness of the linear typing is the fact that the transformation never attempts to form a CPS value or transformed term which depends on a continuation. In other words, the linear zone of the context of any CPS value or transformed term will be empty. Given this, the linearity constraints of the application rules are immediately satisfied since, in the types used by the transformation, the arguments of intuitionistic functions are CPS values, as are the linear functions.

The transformation has this property because source procedures always return to the site from which they were last called, that is, they always return to their return continuations. Further, continuations are not *reified*, or named by source identifiers [FW84]. This ensures that the ability of source procedures to contain free identifiers does not lead to free source identifiers referring to continuations, and hence does not lead to free intuitionistic identifiers referring to continuations in the target. So there are no continuations a procedure needs to know about: it will be given a return continuation and that is enough, and hence the linear zone is empty.

No attempt to discard a continuation is made since every procedure must return by invoking its return continuation, there is simply nothing else for the procedure to do: it cannot loop since the source language is simply-typed and its return continuation is the only jump destination it knows of.

Note that linear use of continuations arises in the target not because of any explicit linearity in the source, but due to the stylized use of the implicit continuations. This is similar to O'Hearn and Reynolds's work [OR00], where linearity and polymorphism arise in the target of a translation from Algol; this prevents the state from being treated, semantically, as if it were first-class.

Proof

1. By structural induction on the source derivation:

   [ID]: The source derivation is of form

$$\frac{}{\Gamma\,, x : T \vdash x : T}$$

   and the target derivation is immediate:

$$\frac{\dfrac{\overline{\overline{\Gamma}\,, x : \overline{T}\,;\, k : \neg\overline{T} \vdash k : \neg\overline{T}} \qquad \overline{\overline{\Gamma}\,, x : \overline{T}\,;\, - \vdash x : \overline{T}}}{\overline{\Gamma}\,, x : \overline{T}\,;\, k : \neg\overline{T} \vdash k\,x : \mathbf{R}}}{\overline{\overline{\Gamma}\,, x : \overline{T}\,;\, - \vdash \delta k.\, k\,x : \neg\overline{T} \multimap \mathbf{R}}}$$

   [ABS]: The source derivation is of form

$$\frac{\begin{array}{c}\vdots \\ \Gamma\,, x : S \vdash N : T\end{array}}{\Gamma \vdash \lambda x.\, N : S \to T}$$

   and hence the induction hypothesis ensures

$$\overline{\Gamma}\,, x : \overline{S}\,;\, - \vdash \overline{N} : \neg\overline{T} \multimap \mathbf{R}$$

from which the target derivation follows:

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma}, x:\overline{S}\ ;\ -\vdash \overline{N}:\neg\overline{T}\multimap \mathbf{R} \qquad \overline{\overline{\Gamma}, x:\overline{S}\ ;\ h:\neg\overline{T}\vdash h:\neg\overline{T}}}{\overline{\Gamma}, x:\overline{S}\ ;\ h:\neg\overline{T}\vdash \overline{N}\,h:\mathbf{R}}}{\overline{\overline{\Gamma}\ ;\ h:\neg\overline{T}\vdash \lambda x.\,\overline{N}\,h:\neg\overline{S}}} \qquad \overline{\overline{\Gamma}\ ;\ -\vdash \delta h.\,\lambda x.\,\overline{N}\,h:\neg\overline{T}\multimap\neg\overline{S}}}{\dfrac{\overline{\overline{\Gamma}\ ;\ k:\neg\overline{S\to T}\vdash k:\neg\overline{S\to T}} \qquad \overline{\overline{\Gamma}\ ;\ -\vdash \delta h.\,\lambda x.\,\overline{N}\,h:\neg\overline{T}\multimap\neg\overline{S}}}{\overline{\Gamma}\ ;\ k:\neg\overline{S\to T}\vdash k\,\delta h.\,\lambda x.\,\overline{N}\,h:\mathbf{R}}}{\overline{\overline{\Gamma}\ ;\ -\vdash \delta k.\,k\,\delta h.\,\lambda x.\,\overline{N}\,h:\neg\overline{S\to T}\multimap\mathbf{R}}}$$

[App]: The source derivation is of form

$$\dfrac{\Gamma\vdash N:S\to T \qquad \Gamma\vdash O:S}{\Gamma\vdash N\,O:T}$$

and hence both

$$\overline{\Gamma}\ ;\ -\vdash \overline{N}:\neg\overline{S\to T}\multimap\mathbf{R}$$

and

$$\overline{\Gamma}\ ;\ -\vdash \overline{O}:\neg\overline{S}\multimap\mathbf{R}$$

are ensured by the induction hypothesis. Let $\Gamma' = \overline{\Gamma}, m:\overline{S\to T}$ and $\Gamma'' = \Gamma', n:\overline{S}$, and the target derivation is

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma''\ ;\ -\vdash m:\neg\overline{T}\multimap\neg\overline{S}} \qquad \overline{\Gamma''\ ;\ k:\neg\overline{T}\vdash k:\neg\overline{T}}}{\Gamma''\ ;\ k:\neg\overline{T}\vdash m\,k:\neg\overline{S}} \qquad \overline{\Gamma''\ ;\ -\vdash n:\overline{S}}}{\Gamma''\ ;\ k:\neg\overline{T}\vdash m\,k\,n:\mathbf{R}}}{\Gamma'\ ;\ k:\neg\overline{T}\vdash \overline{O}\,\lambda n.\,m\,k\,n:\mathbf{R}} \qquad \overline{\Gamma'\ ;\ k:\neg\overline{T}\vdash \lambda n.\,m\,k\,n:\neg\overline{S}}}{\overline{\Gamma}\ ;\ k:\neg\overline{T}\vdash \overline{N}\,\lambda m.\,\overline{O}\,\lambda n.\,m\,k\,n:\mathbf{R}} \qquad \overline{\overline{\Gamma}\ ;\ k:\neg\overline{T}\vdash \lambda m.\,\overline{O}\,\lambda n.\,m\,k\,n:\neg\overline{S\to T}}}{\overline{\overline{\Gamma}\ ;\ -\vdash \delta k.\,\overline{N}\,\lambda m.\,\overline{O}\,\lambda n.\,m\,k\,n:\neg\overline{T}\multimap\mathbf{R}}}$$

2. Immediate. □

2.5 Conclusion

Simply-typed, call-by-value, λ-calculus provides a context in which to see the basic idea of refining a continuation semantics by forcing linear use of continuations with a minimum of fuss. After recalling the λ-calculus, and introducing the core of the linear target language, we have seen that the linear type system is permissive enough to allow restricting the return continuations in the standard CPS transformation to linear usage:

$$\overline{S\to T} \stackrel{\text{def}}{=} \underbrace{(\overline{T}\to\mathbf{R})}_{\substack{\text{return}\\ \text{continuation}}} \multimap \underbrace{\overline{S}\to\mathbf{R}}_{\substack{\text{call}\\ \text{continuation}}}$$

Intuitively, this is because once a procedure returns to a call site, that site will not be returned to again; and once a procedure is called, it will return (or diverge). And technically, because the transformation never attempts to form a CPS value or transformed term which depends on a continuation.

Chapter 3

Procedures, Untyped

In Chapter 2 we gave a treatment of simply-typed λ-calculus which demonstrated the idea of linear use of control contexts. However, the typed nature of the source language has very little impact on control behavior, so we add recursion to the target language and give a very similar interpretation of untyped λ-calculus. The primary conceptual goal is to explain how linearity is happy with recursion. In addition to soundness, in this chapter we consider a form of completeness. This shows how control behavior not present in the source language is disallowed by linearity in the target language, ensuring that linear use of control contexts does indeed constrain the source language and interpretation.

3.1 Source Language

We again consider call-by-value, left-to-right λ-calculus—the source language of Chapter 2—but now without the typing discipline. As before, this language is entirely standard and so we only briefly summarize the formulation we use.

3.1.1 Syntax

In order to state the operational semantics, we reformulate the syntax making the distinction between value terms and general terms explicit in the grammar:

$$\begin{array}{rcll} V & ::= & & \textit{values} \\ & & \mid x & \text{identifier} \\ & & \mid \lambda x.\, M & \text{abstraction} \\ M & ::= & & \textit{terms} \\ & & \mid V & \text{value} \\ & & \mid M\, M & \text{application} \end{array}$$

As an example of the sort of behavior possible with the lifting of the simple typing discipline, consider the following (tail-recursive) program for computing 5 factorial:

$$\begin{array}{l} (\lambda \mathit{fact}.\, \mathit{fact}\, 5) \\ \Big(\big(\lambda Y.\, \lambda n.\, Y\, (\lambda \mathit{loop}.\, \lambda n.\, \lambda a.\, n \to a \,[\!]\, \mathit{loop}\, n{-}1\, a{*}n)\, n\, 1 \big) \\ \quad \big((\lambda Z.\, Z\, Z)\, (\lambda z.\, \lambda f.\, f\, \lambda x.\, \lambda y.\, z\, z\, f\, x\, y) \big) \Big) \end{array} \tag{3.1}$$

Here we have used integers and a conditional if-zero construct ($(\cdot) \to (\cdot) \,[\!]\, (\cdot)$) instead of Church encodings for legibility. The point to note is that recursion is not primitive in the language, instead the recursive procedure is defined using self-application via Y, a Church-style call-by-value fixed-point combinator for two arguments.

3.1.2 Operational semantics

The big-step operational semantics is given by the judgement $M \Downarrow V$ which states that term M evaluates to value V:

$$[\textsc{Val}]\ \frac{}{V \Downarrow V} \qquad\qquad [\textsc{Abs}]\ \frac{M_0 \Downarrow \lambda x.\, M_2 \quad M_1 \Downarrow V_1 \quad M_2[x \mapsto V_0] \Downarrow V_0}{M_0\, M_1 \Downarrow V_0}$$

3.2 Target Language $(+= X + \mu X.\, (\cdot))$

The target language is extended with recursive types to accommodate the recursion implicit in the untyped source language. We will also need to make reference to the target language's equational theory when considering adequacy.

3.2.1 Equational theory

The equational theory is generated in the usual way from the axioms

$$\frac{}{(\lambda x.\, M)\, N \overset{\beta\eta}{=} M[x \mapsto N]} \qquad \frac{}{\lambda x.\, M\, x \overset{\beta\eta}{=} M}\ x \notin \mathsf{fi}(M)$$

$$\frac{}{(\delta x.\, M) \llcorner N \overset{\beta\eta}{=} M[x \mapsto N]} \qquad \frac{}{\delta x.\, M \llcorner x \overset{\beta\eta}{=} M}\ x \notin \mathsf{fi}(M)$$

Note that the full β and η laws hold in the target language, despite the call-by-value evaluation order of the source language, since CPS is evaluation order independent [Mor93, Plo75].

3.2.2 Recursive types

We extend the grammar of types of Section 2.2 with productions for type identifiers and recursive types, and reformulate the grammar to make the distinction between type metaidentifiers A and P, which we will discuss shortly:

$P ::=$		*pointed types*
	$\mid \mathbf{R}$	result type
	$\mid A \to P$	function type
	$\mid P \multimap P$	restricted function type
	$\mid X$	type identifier
	$\mid \mu X.\, P$	recursive type
$A ::=$		*types*
	$\mid \mathbf{N}$	base type
	$\mid A \to A$	nonrecursive function type
	$\mid P$	pointed type

Convention: As in an abstraction term, the body of a recursive type extends as far to the right as possible, so $\mu X.\, P \multimap Q$ parses to $\mu X.\, (P \multimap Q)$ rather than $(\mu X.\, P) \multimap Q$.

Types P are pointed while types A are not necessarily. Pointed types are those for which recursion is allowed. In particular, primitive types used to treat atomic data (such as $\mathbf{N}$) should not be pointed in a CPS language. This is because the result type is, as discussed in Chapter 1, an abstraction of a program's computational effects. Thus, effects occur only at type $\mathbf{R}$, simplifying the interpretation of the other type constructors; For instance, call-by-name and call-by-value coincide. With the addition of recursion comes divergence, which should not occur at type $\mathbf{N}$. At times it is useful to have inert values of function type, so we include a type of nonrecursive functions $A \to A$.

The inclusion of recursive types, and hence type identifiers, makes writing syntactically correct but meaningless types possible, so some extra machinery is required to ensure the types we use are meaningful. This can be done (following [AF96]) by defining the usual equality on types and extending the typing rules of Section 2.2 with a formal version of

$$[\textsc{Rec}]\ \frac{\Gamma\ ;\ \Delta \vdash M : B}{\Gamma\ ;\ \Delta \vdash M : A}\ B = A$$

The details are standard and presented in Section A.3.

3.3 Refined CPS Transformation

Since the source language is untyped, the form of the CPS transformation is slightly different. Previously, source types were transformed to target types. But now there are no source types and so we define a single target type D which will interpret all source values. Similarly, source contexts are not transformed; instead all intuitionistic identifiers in the target will have type D. And so instead of transforming source judgments to target judgments, we now transform each source term

$$M$$

to the target judgment

$$x_1 : D\ ,\ \ldots\ ,\ x_n : D\ ;\ -\ \vdash \overline{M} : (D \to \mathbf{R}) \multimap \mathbf{R}$$

where $\{x_1, \ldots, x_n\} \supseteq \mathsf{fi}(M)$ for some $n \geq 0$.

The driving aspect of the transformation is the type used to interpret value terms (procedures)

$$D \stackrel{\text{def}}{=} \mu D.\quad \underbrace{(D \to \mathbf{R})}_{\substack{\text{return}\\ \text{continuation}}}\ \multimap\ \underbrace{D \to \mathbf{R}}_{\substack{\text{call}\\ \text{continuation}}} \tag{3.2}$$

which effectively defines D to be the least solution of the domain isomorphism

$$D \cong (D \to \mathbf{R}) \multimap D \to \mathbf{R}$$

The type D corresponds extremely closely to the transformation of the procedure type in Chapter 2. We no longer say anything about the types of the data which is passed, but the handling of continuations is unchanged. With this interpretation, the transformation on terms, given in Figure 3.1, is equivalent to that in Chapter 2, but we reformulate it since the syntactic categories of values and terms are now distinct, although their transformations are mutually recursive. The target terms are identical in the typed and untyped transformations, it is only the

Figure 3.1 Refined CPS of Untyped λ-calculus

Value Terms

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to \mathbf{R})}_{\text{return}} \multimap \underbrace{D \to \mathbf{R}}_{\text{call}}$$

$$\ulcorner x \urcorner \stackrel{\text{def}}{=} x$$
$$\ulcorner \lambda x.\, M \urcorner \stackrel{\text{def}}{=} \delta k.\ \lambda x.\, \overline{M} \llcorner k$$

Terms

$$\underbrace{(D \to \mathbf{R})}_{\text{return}} \multimap \mathbf{R}$$

$$\overline{V} \stackrel{\text{def}}{=} \delta k.\, k \ulcorner V \urcorner$$
$$\overline{M\,N} \stackrel{\text{def}}{=} \delta k.\, \overline{M} \llcorner \lambda m.\, \overline{N} \llcorner \lambda n.\, m \llcorner k\, n \quad m \notin \mathsf{fi}(N)$$

Programs

$$\underbrace{(D \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{R}$$

$$\llcorner M \lrcorner \stackrel{\text{def}}{=} \overline{M}$$

types in the target judgments which are different. Hence the explanations of the intuitions behind this transformation carry over directly from Chapter 2.

The interpretation of programs using type[1]

$$\underbrace{(D \to \mathbf{R})}_{\substack{\text{toplevel} \\ \text{continuation}}} \multimap \mathbf{R}$$

is trivial, see Figure 3.1.

3.4 Soundness

Now that we distinguish between values and terms, the soundness of the transformation of each depends on the other.

[1] Instead of the type D we define, we really ought to instead use a type such as

$$\mu D.\, \mathbf{N} + ((D \to \mathbf{R}) \multimap D \to \mathbf{R})$$

This would allow us to give an interpretation of toplevel programs using the type

$$(\mathbf{N} \to \mathbf{R}) \multimap \mathbf{R}$$

as in Chapter 2, which isolates the operating system from the interpretation of control behavior of the source language. As it stands, the operating system has to know about how we handle procedures in order to extract any information from the value passed to the toplevel continuation. But using such a type would significantly complicate the presentation of the transformation for very little conceptual gain.

Proposition 2 (Soundness) *For any source term M, if* $\{x_1, \ldots, x_n\} \supseteq \mathrm{fi}(M)$ *for some* $n \geq 0$*, then*

1. $$x_1 : D\,, \ldots, x_n : D\,;\, - \vdash \overline{M} : (D \to \mathbf{R}) \multimap \mathbf{R}$$

   and, if M is a value, then

   $$x_1 : D\,, \ldots, x_n : D\,;\, - \vdash \ulcorner M \urcorner : D$$

2. $$x_1 : D\,, \ldots, x_n : D\,;\, - \vdash \llcorner M \lrcorner : (D \to \mathbf{R}) \multimap \mathbf{R}$$

The soundness proof is straightforward, perhaps deceptively so, since it does not address the relationship between recursion and linearity very explicitly. The key point is that continuations are not themselves recursive in this interpretation. If they were, then invoking a continuation would eventually result in the continuation being invoked again, recursively, which would break the linear typing.[2] Instead, continuation transformers are recursive, which is not problematic since the interpretation does not constrain their use. Concretely, the transformation of the self-application of an identifier (such as that in (3.1))

$$\overline{f\ f} \overset{\beta\eta}{=} \delta k.\ f \llcorner k \lrcorner\ f$$

shows that, in the target language, recursion (even tail-) is effected by a sort of self-application in which a continuation transformer f is passed to a continuation $f \llcorner k \lrcorner$ which is obtained from f itself. If we were to uncurry the type of continuation transformers, a call to f would directly pass itself as one of the arguments. The important point here is that self-application in the source language does not imply that continuation transformers are nonlinear functions; that is, it is entirely possible for the continuation $f \llcorner k \lrcorner$ to be a nonlinear function, without forcing f to be a nonlinear function. The typing derivation of self-application in the target language

$$\dfrac{\dfrac{\dfrac{f : D\,;\, - \vdash f : D}{f : D\,;\, - \vdash f : \neg D \multimap \neg D}\, D = \neg D \multimap \neg D \qquad \dfrac{}{f : D\,;\, k : \neg D \vdash k : \neg D}}{f : D\,;\, k : \neg D \vdash f \llcorner k \lrcorner : \neg D} \qquad \dfrac{}{f : D\,;\, - \vdash f : D}}{\dfrac{f : D\,;\, k : \neg D \vdash f \llcorner k \lrcorner\ f : \mathbf{R}}{f : D\,;\, - \vdash \delta k.\ f \llcorner k \lrcorner\ f : \neg D \multimap \mathbf{R}}}$$

shows how the recursive type must be unfolded once to type the operand occurrence of f, reinforcing that it is continuation transformers, not continuations, which are recursive.

On the other hand, recursion allows a term to diverge without ever invoking its return continuation, but this in no way violates linear typing. To understand this, it is essential to realize that a continuation which is "used linearly" is not necessarily "invoked exactly once," nor vice versa. Here, "use" is referring to the static type system, while "invoke" is referring to the dynamic operational behavior, and the relationship between the two is not immediately obvious. Linear typing can only make statements about dynamic behavior indirectly. Linear typing makes direct constraints at the level of environment handling, which then imply properties about the dynamic behavior, but the indirectness of this connection thwarts many intuitions about the dynamic behavior of linearly typed code. So the non-discarding property of linear typing roughly means not that every continuation will be invoked, but that no continuation will become garbage.

[2] Later, in Section 8.2.2, we will attempt to define continuations recursively, and will witness the breakage of linearity concretely.

In this light, the ability of a program to diverge without ever invoking its return continuation is unproblematic.

Proof

1. Let $\Gamma = x_1 : D\ , \ldots , x_n : D$ and proceed by structural induction on the syntax of M:

   $[V]$: Therefore the induction hypothesis ensures

   $$\Gamma\ ;\ -\ \vdash\ \ulcorner V \urcorner : D$$

   from which the target derivation follows:

   $$\dfrac{\dfrac{}{\Gamma\ ;\ k : \neg D \vdash k : \neg D} \qquad \Gamma\ ;\ -\ \vdash\ \ulcorner V \urcorner : D}{\dfrac{\Gamma\ ;\ k : \neg D \vdash k\ \ulcorner V \urcorner : \mathbf{R}}{\Gamma\ ;\ -\ \vdash\ \delta k.\, k\ \ulcorner V \urcorner : \neg D \multimap \mathbf{R}}}$$

   $[N\,O]$: Therefore the induction hypothesis ensures

   $$\Gamma\ ;\ -\ \vdash\ \overline{N} : \neg D \multimap \mathbf{R}$$

   and

   $$\Gamma\ ;\ -\ \vdash\ \overline{O} : \neg D \multimap \mathbf{R}$$

   Let $\Gamma' = \Gamma\ , m : D\ , n : D$, and the target derivation is

   $$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma'\ ;\ -\ \vdash\ m : D}}{\Gamma'\ ;\ -\ \vdash\ m : \neg D \multimap \neg D}\ D = \neg D \multimap \neg D \qquad \dfrac{}{\Gamma'\ ;\ k : \neg D \vdash k : \neg D}}{\Gamma'\ ;\ k : \neg D \vdash m \llcorner k : \neg D} \qquad \dfrac{}{\Gamma'\ ;\ -\ \vdash\ n : D}}{\Gamma'\ ;\ k : \neg D \vdash m \llcorner k\, n : \mathbf{R}}}{\Gamma\ , m : D\ ;\ -\ \vdash\ \overline{O} : \neg D \multimap \mathbf{R} \qquad \dfrac{\Gamma\ , m : D\ ;\ k : \neg D \vdash \lambda n.\, m \llcorner k\, n : \neg D}{\Gamma\ , m : D\ ;\ k : \neg D \vdash \overline{O} \llcorner \lambda n.\, m \llcorner k\, n : \mathbf{R}}}}{\dfrac{\Gamma\ ;\ -\ \vdash\ \overline{N} : \neg D \multimap \mathbf{R} \qquad \dfrac{\Gamma\ ;\ k : \neg D \vdash \lambda m.\, \overline{O} \llcorner \lambda n.\, m \llcorner k\, n : \neg D}{\Gamma\ ;\ k : \neg D \vdash \overline{N} \llcorner \lambda m.\, \overline{O} \llcorner \lambda n.\, m \llcorner k\, n : \mathbf{R}}}{\Gamma\ ;\ -\ \vdash\ \delta k.\, \overline{N} \llcorner \lambda m.\, \overline{O} \llcorner \lambda n.\, m \llcorner k\, n : \neg D \multimap \mathbf{R}}}$$

   Note that D is unfolded once.

Now suppose $M = V$ for some value V and proceed by cases on the syntax of V:

$[x]$: $x = x_i$ for $1 \leq i \leq n$ and the target derivation is immediate:

$$\dfrac{}{\Gamma\ ;\ -\ \vdash\ x : D}$$

$[\lambda x.\, N]$: Therefore the induction hypothesis ensures

$$\Gamma\ , x : D\ ;\ -\ \vdash\ \overline{N} : \neg D \multimap \mathbf{R}$$

from which the target derivation follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma\,, x : D\,;\, - \vdash \overline{N} : \neg D \multimap \mathbf{R} \qquad \overline{\Gamma\,, x : D\,;\, k : \neg D \vdash k : \neg D}}{\Gamma\,, x : D\,;\, k : \neg D \vdash \overline{N} \llcorner k : \mathbf{R}}}{\Gamma\,;\, k : \neg D \vdash \lambda x.\, \overline{N} \llcorner k : \neg D}}{\Gamma\,;\, - \vdash \delta k.\, \lambda x.\, \overline{N} \llcorner k : \neg D \multimap \neg D}}{\Gamma\,;\, - \vdash \delta k.\, \lambda x.\, \overline{N} \llcorner k : D}\ \neg D \multimap \neg D = D$$

Note that D is folded once.

2. Immediate. □

3.5 Adequacy

Since the CPS transformation is standard but for typing, the computational adequacy, that is, the correctness of the computed results, is standard.

Proposition 3 (Adequacy) *For any closed source term M and any target term K, $M \Downarrow V$ if and only if $\overline{M} \llcorner K \stackrel{\beta\eta}{=} K \ulcorner V \urcorner$ and $\lfloor M \rfloor \llcorner K \stackrel{\beta\eta}{=} K \ulcorner V \urcorner$.*

Proof Modulo slight differences in setting, by the proof of Plotkin's Simulation Theorem [Plo75]. □

3.6 Completeness

Proposition 2 establishes the correctness of the refinement of the standard semantics resulting from restricting to linear use of control contexts. We have not yet, however, argued that this refinement is tight in any way. That is, we would like evidence that a discipline of linearly used control contexts in the target forces target terms to exhibit only control behavior which is possible in the source.

An example consequence of the restrictions imposed by the refined interpretation, stated informally, is:

> Once a procedure returns, it will not return again until it is called again.

This control-flow property is captured by making the CPS versions of such procedures inexpressible. More concretely, using the standard (unrefined) version of (3.2)

$$D \stackrel{\text{def}}{=} \mu D.\, (D \to \mathbf{R}) \to D \to \mathbf{R}$$

and recalling the example from Section 1.2, we have in the target

$$- \,;\, - \vdash \lambda r.\, r\, \lambda k.\, \lambda z.\, k\, \lambda h.\, \lambda x.\, k\, x : (D \to \mathbf{R}) \to \mathbf{R}$$

This term is the standard CPS version of ("comes from")

$$\texttt{return/cc} \stackrel{\text{def}}{=} \lambda z.\, \texttt{call/cc}\, \lambda k.\, k$$

and exhibits the backtracking behavior characteristic of first-class continuations. But this control behavior is not expressible in the source language without first-class continuations [Thi99b]. So the standard interpretation of λ-calculus is *junky*, meaning that there are terms (of the appropriate type) in the target language which do not come from any source term.

On the other hand, using the refined interpretation of procedures

$$D \stackrel{\text{def}}{=} \mu D.\, (D \to \mathbf{R}) \multimap D \to \mathbf{R}$$

the judgment

$$-\,;\,- \vdash \delta r.\, r\,\delta k.\, \lambda z.\, k\,\delta h.\, \lambda x.\, k\,x : (D \to \mathbf{R}) \multimap \mathbf{R}$$

is not derivable, primarily because it involves the CPS version of a value, $\delta h.\, \lambda x.\, k\,x$, with a free continuation, k. This is a demonstration of a main way a discipline of linearly used control contexts limits expressive power. The use of values in the source language is not constrained, and so they are interpreted by target types whose use is also unconstrained, and the CPS versions of source values end up being arguments of intuitionistic functions; usually continuations. The linear type system—the [APP] rule in particular—then ensures such arguments do not have any free control context identifiers. So the preclusion of CPS values containing control contexts is a central characteristic of the system.

The source language analogue of the absence of CPS values with free control context identifiers is the absence of reified upward control contexts. In a source language with procedures, a value is *upward* if it is returned or stored [FWH01]. In the presence of other control constructs, this definition must be altered accordingly. For instance, in a language with exceptions, a value raised as an exception is also upward. Conversely, a value is *downward* if it is not upward. So intuitively, if control contexts are both reified and upward in the source language, then there will be CPS values with free control context identifiers, and hence the source language will not admit a discipline of linearly used control contexts.

Rather than argue that there are no junk terms whatsoever, we prove a somewhat less general result:

Proposition 4 (No Junk) *If* $\Gamma\,;\,- \models_{\mathrm{s}} M : (D \to \mathbf{R}) \multimap \mathbf{R}$, *then there exists a source term* N *such that* $M \stackrel{\beta\eta}{=} \overline{N}$.

(Note that $\stackrel{\beta\eta}{=}$ is a very fine equality, and hence similar properties will hold of coarser (contextual) equivalence relations.)

This result is not fully general due to the use of a specialized typing judgment $(\cdot)\,;\,(\cdot) \models_{\mathrm{s}} (\cdot) : (\cdot)$ rather than the full $(\cdot)\,;\,(\cdot) \vdash (\cdot) : (\cdot)$. However, later work by others has extended this result to the general case, see Section 12.2.

The specialized judgment characterizes the range, not the codomain, of the CPS transformation. This is accomplished by "carving out" a sublanguage of the target by restricting the form of types to those the CPS transformation actually exercises. Subterms of M in $\Gamma\,;\,- \vdash M : (D \to \mathbf{R}) \multimap \mathbf{R}$ may be very "non-CPS" or may exhibit essentially unconstrained control behavior in the process of producing a term of an appropriate type. In other words, $\Gamma\,;\,- \vdash M : (D \to \mathbf{R}) \multimap \mathbf{R}$ constrains M to use types—and hence exhibit behavior—appropriate to procedures at toplevel but its subterms are not so constrained, while $\Gamma\,;\,- \models_{\mathrm{s}} M : (D \to \mathbf{R}) \multimap \mathbf{R}$ constrains M and its subterms equally. In this way we rule out not only essentially unconstrained control behaviors, but also control behaviors such as exceptions (Chapter 4) or coroutines (Chapter 9) which are interpretable using control contexts linearly but with different types than those used to interpret procedures. Intuitively, this specialization of types fixes the form of control contexts, and Proposition 4 then asserts that using these control contexts linearly, no control behavior beyond that expressible with procedures is possible. Behavior arising from exceptions or coroutines, say, is precluded since they require control contexts of different form, but this is not the case for first-class continuations, whose behavior is ruled out by linearity.

The carved sublanguage is more like a compiler intermediate language than is the full target language since programs in the difference are likely to cause later compilation stages to be unpleasantly surprised, to say the least. So, from an implementation point of view it is not at all obvious that the more general completeness result has any value beyond the specialized one.

Note that while Proposition 4 ensures that the control behavior expressible in the carved sublanguage is not more than that expressible in the source, it does not imply the impossibility of an interpretation of a language with first-class continuations, say, which uses control contexts linearly. While we strongly believe that a conceptually reasonable interpretation of first-class continuations which uses control contexts linearly does not exist, we are not optimistic that a satisfactory theoretical analysis of this is possible. Attacking this problem would lead into an expressiveness tar pit, due to Gödel encodings and so on. Defining what qualifies as a "good interpretation" would be a very significant task, and though one can learn much from such attempts [BL96, Fel91, Lan98], we have not yet seen one which is exempt from criticism. So the question of the existence of an interpretation of first-class continuations would change to the question of the existence of a characterization of "good interpretation" which allows an interpretation of first-class continuations.

3.6.1 Target sublanguage

With the overall plan laid, we proceed to define the target sublanguage by restricting the form of types to only those used by the CPS transformation.

The syntax of terms is unchanged from that in Section 2.2.1, and the equational theory is unchanged from that in Section 3.2.1. Since the interpretation in Section 3.3 revolves around the type

$$D \stackrel{\text{def}}{=} \mu D.\, (D \to \mathbf{R}) \multimap D \to \mathbf{R}$$

the syntax of types is built from it:

$$\begin{array}{lll} A, P ::= & & \textit{types} \\ & \mid D \to \mathbf{R} & \text{continuation type} \\ & \mid D \mid (D \to \mathbf{R}) \multimap D \to \mathbf{R} & \text{continuation transformer types} \\ & \mid (D \to \mathbf{R}) \multimap \mathbf{R} & \text{“program” type} \\ & \mid \mathbf{R} & \text{result type} \end{array} \tag{3.3}$$

Note that there is only a single type identifier, and only a single unfolding of the recursive continuation transformer type is needed, or allowed.

The typing rules of Section 3.2.2 specialized to the restricted form of types are

$$\frac{}{\Gamma\,;\,\Delta\,,\,k : \neg D \vDash_{\mathrm{s}} k : \neg D} \qquad \frac{\Gamma\,,\,x : D\,;\,\Delta \vDash_{\mathrm{s}} M : \mathbf{R}}{\Gamma\,;\,\Delta \vDash_{\mathrm{s}} \lambda x.\, M : \neg D}$$

$$\frac{\Gamma\,;\,\Delta \vDash_{\mathrm{s}} M : \neg D \multimap \neg D \quad \Gamma\,;\,\Delta' \vDash_{\mathrm{s}} N : \neg D}{\Gamma\,;\,\Delta\,,\,\Delta' \vDash_{\mathrm{s}} M \llcorner N : \neg D}$$

$$\frac{}{\Gamma\,,\,x : D\,;\,\Delta \vDash_{\mathrm{s}} x : D} \qquad \frac{\Gamma\,;\,\Delta\,,\,k : \neg D \vDash_{\mathrm{s}} M : \neg D}{\Gamma\,;\,\Delta \vDash_{\mathrm{s}} \delta k.\, M : \neg D \multimap \neg D}$$

$$\frac{\Gamma\,;\,\Delta \vDash_{\mathrm{s}} M : \neg D \multimap \neg D}{\Gamma\,;\,\Delta \vDash_{\mathrm{s}} M : D} \qquad \frac{\Gamma\,;\,\Delta \vDash_{\mathrm{s}} M : D}{\Gamma\,;\,\Delta \vDash_{\mathrm{s}} M : \neg D \multimap \neg D}$$

$$\frac{\Gamma\,;\,\Delta\,,\,k : \neg D \vDash_{\mathrm{s}} M : \mathbf{R}}{\Gamma\,;\,\Delta \vDash_{\mathrm{s}} \delta k.\, M : \neg D \multimap \mathbf{R}}$$

$$\frac{\Gamma\,;\,\Delta \vDash_{\mathrm{s}} M : \neg D \quad \Gamma\,;\,- \vDash_{\mathrm{s}} N : D}{\Gamma\,;\,\Delta \vDash_{\mathrm{s}} M\,N : \mathbf{R}} \qquad \frac{\Gamma\,;\,\Delta \vDash_{\mathrm{s}} M : \neg D \multimap \mathbf{R} \quad \Gamma\,;\,\Delta' \vDash_{\mathrm{s}} N : \neg D}{\Gamma\,;\,\Delta\,,\,\Delta' \vDash_{\mathrm{s}} M \llcorner N : \mathbf{R}}$$

Observation 5 $\Gamma\ ;\ \Delta \models_{\mathrm{s}} M : A$ *if and only if* $\Gamma\ ;\ \Delta \vdash M : A$ *and all types which appear in the derivation of the latter judgment are in the type language defined by (3.3).*

Note that all the derivations we have presented so far satisfy this property.

Observation 6 *If* $\Gamma\ ;\ \Delta \models_{\mathrm{s}} M : A$*, then* $\Delta = -$ *or* $\Delta = k : \neg D$*.*

Note that this indicates that with the restricted form of types, affine, linear, and ordered linear typing all coincide.[3]

Taking this special form of contexts into account, we can further specialize the presentation of the typing rules:

$$[\textsc{KId}]\ \frac{}{\Gamma\ ;\ k : \neg D \models_{\mathrm{s}} k : \neg D} \qquad [\textsc{KAbs}]\ \frac{\Gamma\ ,\ x : D\ ;\ k : \neg D \models_{\mathrm{s}} M : \mathbf{R}}{\Gamma\ ;\ k : \neg D \models_{\mathrm{s}} \lambda x.\, M : \neg D}$$

$$[\textsc{KApp}]\ \frac{\Gamma\ ;\ - \models_{\mathrm{s}} M : \neg D \multimap \neg D \quad \Gamma\ ;\ k : \neg D \models_{\mathrm{s}} N : \neg D}{\Gamma\ ;\ k : \neg D \models_{\mathrm{s}} M \_ N : \neg D}$$

$$[\textsc{DId}]\ \frac{}{\Gamma\ ,\ x : D\ ;\ - \models_{\mathrm{s}} x : D} \qquad [\textsc{DAbs}]\ \frac{\Gamma\ ;\ k : \neg D \models_{\mathrm{s}} M : \neg D}{\Gamma\ ;\ - \models_{\mathrm{s}} \delta k.\, M : \neg D \multimap \neg D}$$

$$[\textsc{DFold}]\ \frac{\Gamma\ ;\ - \models_{\mathrm{s}} M : \neg D \multimap \neg D}{\Gamma\ ;\ - \models_{\mathrm{s}} M : D} \qquad [\textsc{DUnFold}]\ \frac{\Gamma\ ;\ - \models_{\mathrm{s}} M : D}{\Gamma\ ;\ - \models_{\mathrm{s}} M : \neg D \multimap \neg D}$$

$$[\textsc{PAbs}]\ \frac{\Gamma\ ;\ k : \neg D \models_{\mathrm{s}} M : \mathbf{R}}{\Gamma\ ;\ - \models_{\mathrm{s}} \delta k.\, M : \neg D \multimap \mathbf{R}}$$

$$[\textsc{RAppK}]\ \frac{\Gamma\ ;\ k : \neg D \models_{\mathrm{s}} M : \neg D \quad \Gamma\ ;\ - \models_{\mathrm{s}} N : D}{\Gamma\ ;\ k : \neg D \models_{\mathrm{s}} M\, N : \mathbf{R}}$$

$$[\textsc{RAppP}]\ \frac{\Gamma\ ;\ - \models_{\mathrm{s}} M : \neg D \multimap \mathbf{R} \quad \Gamma\ ;\ k : \neg D \models_{\mathrm{s}} N : \neg D}{\Gamma\ ;\ k : \neg D \models_{\mathrm{s}} M \_ N : \mathbf{R}}$$

3.6.2 Extracting the range of CPS

As an aside, from this specialized type system we can extract a grammar where we have one syntactic category for terms of each type:

$$\begin{array}{lll} K ::= k \mid \lambda x.\, R \mid D \_ K & & \neg D \\ D ::= x \mid \delta k.\, K & & D = \neg D \multimap \neg D \\ P ::= \delta k.\, R & & \neg D \multimap \mathbf{R} \\ R ::= K\, D \mid P \_ K & & \mathbf{R} \end{array}$$

If we draw k from a singleton set of identifiers which is disjoint from the set of identifiers from which x is drawn, then the type system and syntax are equivalent. That is, a term will be an element of a syntactic category if and only if there is a judgment proving the term has the associated type.

[3] Note, however, that while we only consider continuation usage, Polakow and Pfenning [PP00] also distinguish continuation arguments from function arguments, and refine the treatment of the former using an ordered system. An unordered system such as ours will not suffice for such analyses.

Sabry and Felleisen's analysis of the range of CPS [SF93] led them to nearly the same syntax. The only difference is that their transformation does not introduce "administrative" redexes [Plo75], and so syntactic category P is not needed, but toplevel programs must be treated specially. This is interesting since Sabry and Felleisen analyzed the syntax of the output of the CPS transformation, while we have analyzed the types of the output of the CPS transformation, obtaining the same result (modulo "administrative" redexes). So this work can be seen as a logical reconstruction of (part of) Sabry and Felleisen's: their methods are syntactic, or even lexical (one continuation identifier is enough), while our starting point is a domain isomorphism with, crucially, a restricted (affine or linear) function space constraining continuation usage.

3.6.3 DS transformation

The proof of Proposition 4 is essentially given by a DS transformation. The DS transformation takes terms in the carved-out target sublanguage to the source language and is, in a sense, inverse to the CPS transformation.

The transformation of programs $\mathcal{P}(\!|(\cdot)|\!)$ is defined mutually recursively with $\mathcal{K}(\!|(\cdot)|\!)$, $\mathcal{D}(\!|(\cdot)|\!)$, and $\mathcal{R}(\!|(\cdot)|\!)$, which transform continuations, continuation transformers, and results, respectively:

$$\begin{aligned}
\mathcal{K}(\!|k|\!) &\stackrel{\text{def}}{=} \lambda x.\, x \\
\mathcal{K}(\!|\lambda x.\, M|\!) &\stackrel{\text{def}}{=} \lambda x.\, \mathcal{R}(\!|M|\!) \\
\mathcal{K}(\!|M \lrcorner N|\!) &\stackrel{\text{def}}{=} \lambda x.\, \mathcal{K}(\!|N|\!)\, (\mathcal{D}(\!|M|\!)\, x) \quad x \notin \mathsf{fi}(M \lrcorner N)
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}(\!|x|\!) &\stackrel{\text{def}}{=} x \\
\mathcal{D}(\!|\delta k.\, M|\!) &\stackrel{\text{def}}{=} \mathcal{K}(\!|M|\!)
\end{aligned}$$

$$\mathcal{P}(\!|\delta k.\, M|\!) \stackrel{\text{def}}{=} \mathcal{R}(\!|M|\!)$$

$$\begin{aligned}
\mathcal{R}(\!|M\, N|\!) &\stackrel{\text{def}}{=} \mathcal{K}(\!|M|\!)\, \mathcal{D}(\!|N|\!) \\
\mathcal{R}(\!|M \lrcorner N|\!) &\stackrel{\text{def}}{=} \mathcal{K}(\!|N|\!)\, \mathcal{P}(\!|M|\!)
\end{aligned}$$

For comparison, if we modify Sabry and Felleisen's DS transformation by allowing the introduction of administrative redexes and eliminating the use of substitution, we obtain the same transformation except that Sabry and Felleisen's does not include clauses for $\mathcal{P}(\!|\delta k.\, M|\!)$ or $\mathcal{R}(\!|M \lrcorner N|\!)$ since their language does not include administrative redexes and they handle toplevel programs specially.

To understand the idea behind this transformation, note that continuations accept an argument, do some computation converting the argument into another value, and then invoke their parent continuation with the new value. So each continuation can be characterized by the function from values to values sitting inside it, together with its parent continuation. The driving idea of the DS transformation is that a continuation is transformed into this function and instead of continuations referring to their parent continuations by name (a continuation depends on exactly one other continuation, Observation 6), the DS transforms of continuations are just composed. For example, in the continuation

$$(\delta k.\, M) \lrcorner N \stackrel{\beta\eta}{=} M[k \mapsto N]$$

M is a continuation which refers to its parent continuation N using the name k. When this continuation is invoked with an argument, M will convert the argument into another value and then invoke N with it. In DS, we have

$$\mathcal{K}(\!|(\delta k.\, M) \lrcorner N|\!) = \lambda x.\, \mathcal{K}(\!|N|\!)\, (\mathcal{K}(\!|M|\!)\, x)$$

from which we see that $\mathcal{K}(\!|M|\!)$ will accept argument x and then convert it into another value; but instead of explicitly invoking the parent continuation, $\mathcal{K}(\!|M|\!)\ x$ returns to the argument position of an application of the transformation of the parent continuation. In other words, explicit manipulation of control contexts has been replaced by introduction of implicit control contexts, simply undoing the effect of the CPS transformation.

3.6.4 No junk

The DS transformation is useful (and correct) only because it is inverse to the CPS transformation.

Lemma 7 (Inverseness) *1. If* $\Gamma\ ;\ k : \neg D \models_{\mathrm{s}} M : \neg D$, *then* $\overline{\mathcal{K}(\!|M|\!)}\llcorner N \overset{\beta\eta}{=} N\ \delta k.\ M$.

2. If $\Gamma\ ;\ - \models_{\mathrm{s}} M : D$ *or* $\Gamma\ ;\ - \models_{\mathrm{s}} M : \neg D \multimap \neg D$, *then* $\overline{\mathcal{D}(\!|M|\!)}\llcorner N \overset{\beta\eta}{=} N\ M$.

3. If $\Gamma\ ;\ - \models_{\mathrm{s}} M : \neg D \multimap \mathbf{R}$, *then* $\overline{\mathcal{P}(\!|M|\!)} \overset{\beta\eta}{=} M$.

4. If $\Gamma\ ;\ k : \neg D \models_{\mathrm{s}} M : \mathbf{R}$, *then* $\overline{\mathcal{R}(\!|M|\!)} \overset{\beta\eta}{=} \delta k.\ M$.

With this result, no junk is immediate:

Proof [Proposition 4] Let $N = \mathcal{P}(\!|M|\!)$ and the result follows by Lemma 7. □

Proof [Lemma 7] By induction on the derivation of the judgment:

[KID]: Therefore $M = k$. Hence

$$\begin{aligned}
\overline{\mathcal{K}(\!|k|\!)}\llcorner N &= \overline{\lambda x.\ x}\llcorner N \\
&= (\delta k.\ k \ulcorner \lambda x.\ x \urcorner)\llcorner N \\
&\overset{\beta\eta}{=} N \ulcorner \lambda x.\ x \urcorner \\
&= N\ \delta k.\ \lambda x.\ \bar{x}\llcorner k \\
&= N\ \delta k.\ \lambda x.\ (\delta k.\ k \ulcorner x \urcorner)\llcorner k \\
&\overset{\beta\eta}{=} N\ \delta k.\ \lambda x.\ k \ulcorner x \urcorner \\
&= N\ \delta k.\ \lambda x.\ k\ x \\
&\overset{\beta\eta}{=} N\ \delta k.\ k
\end{aligned}$$

[KABS]: Therefore $M = \lambda x.\ M'$ and $\Gamma\ , x : D\ ;\ k : K \models_{\mathrm{s}} M' : \mathbf{R}$. Hence

$$\begin{aligned}
\overline{\mathcal{K}(\!|\lambda x.\ M'|\!)}\llcorner N &= \overline{\lambda x.\ \mathcal{R}(\!|M'|\!)}\llcorner N \\
&= \delta k.\ k \ulcorner \lambda x.\ \mathcal{R}(\!|M'|\!) \urcorner \llcorner N \\
&\overset{\beta\eta}{=} N \ulcorner \lambda x.\ \mathcal{R}(\!|M'|\!) \urcorner \\
&= N\ \delta k.\ \lambda x.\ \overline{\mathcal{R}(\!|M'|\!)}\llcorner k \\
&\overset{\beta\eta}{=} N\ \delta k.\ \lambda x.\ (\delta k.\ M')\llcorner k \quad \text{by the induction hypothesis} \\
&\overset{\beta\eta}{=} N\ \delta k.\ \lambda x.\ M'
\end{aligned}$$

[KAPP]: Therefore $M = M'\llcorner N'$, $\Gamma\ ;\ - \models_{\mathrm{s}} M' : K \multimap K$, and $\Gamma\ ;\ k : K \models_{\mathrm{s}} N' : K$. Hence

$$
\begin{aligned}
&\overline{\mathcal{K}(\!|M' \llcorner N'|\!)} \llcorner N \\
&= \overline{\lambda x.\, \mathcal{K}(\!|N'|\!)\ (\mathcal{D}(\!|M'|\!)\ x)} \llcorner N \\
&= (\delta k.\, k \ulcorner \lambda x.\, \mathcal{K}(\!|N'|\!)\ (\mathcal{D}(\!|M'|\!)\ x) \urcorner) \llcorner N \\
&\overset{\beta\eta}{=} N \ulcorner \lambda x.\, \mathcal{K}(\!|N'|\!)\ (\mathcal{D}(\!|M'|\!)\ x) \urcorner \\
&= N\ \delta k.\, \lambda x.\, \overline{\mathcal{K}(\!|N'|\!)\ (\mathcal{D}(\!|M'|\!)\ x)} \llcorner k \\
&= N\ \delta k.\, \lambda x.\, (\delta k.\, \overline{\mathcal{K}(\!|N'|\!)} \llcorner \lambda m.\, \overline{\mathcal{D}(\!|M'|\!)\ x} \llcorner \lambda n.\, m \llcorner k\ n) \llcorner k \\
&\overset{\beta\eta}{=} N\ \delta k.\, \lambda x.\, \overline{\mathcal{K}(\!|N'|\!)} \llcorner \lambda m.\, \overline{\mathcal{D}(\!|M'|\!)\ x} \llcorner \lambda n.\, m \llcorner k\ n \\
&\overset{\beta\eta}{=} N\ \delta k.\, \lambda x.\, \overline{\mathcal{K}(\!|N'|\!)} \llcorner \lambda m.\, \overline{\mathcal{D}(\!|M'|\!)\ x} \llcorner (m \llcorner k) \\
&\overset{\beta\eta}{=} N\ \delta k.\, \lambda x.\, (\lambda m.\, \overline{\mathcal{D}(\!|M'|\!)\ x} \llcorner (m \llcorner k)) \llcorner \delta k.\, N' && \text{by the induction hypothesis} \\
&\overset{\beta\eta}{=} N\ \delta k.\, \lambda x.\, \overline{\mathcal{D}(\!|M'|\!)\ x} \llcorner ((\delta k.\, N') \llcorner k) \\
&\overset{\beta\eta}{=} N\ \delta k.\, \lambda x.\, \overline{\mathcal{D}(\!|M'|\!)\ x} \llcorner N' \\
&= N\ \delta k.\, \lambda x.\, (\delta k.\, \overline{\mathcal{D}(\!|M'|\!)} \llcorner \lambda m.\, \bar{x} \llcorner \lambda n.\, m \llcorner k\ n) \llcorner N' \\
&\overset{\beta\eta}{=} N\ \delta k.\, \lambda x.\, \overline{\mathcal{D}(\!|M'|\!)} \llcorner \lambda m.\, \bar{x} \llcorner \lambda n.\, m \llcorner N'\ n \\
&\overset{\beta\eta}{=} N\ \delta k.\, \lambda x.\, \overline{\mathcal{D}(\!|M'|\!)} \llcorner \lambda m.\, \bar{x} \llcorner (m \llcorner N') \\
&\overset{\beta\eta}{=} N\ \delta k.\, \lambda x.\, (\lambda m.\, \bar{x} \llcorner (m \llcorner N'))\ M' && \text{by the induction hypothesis} \\
&\overset{\beta\eta}{=} N\ \delta k.\, \lambda x.\, \bar{x} \llcorner (M' \llcorner N') \\
&= N\ \delta k.\, \lambda x.\, (\delta k.\, k \ulcorner x \urcorner) \llcorner (M' \llcorner N') \\
&\overset{\beta\eta}{=} N\ \delta k.\, \lambda x.\, M' \llcorner N' \ulcorner x \urcorner \\
&= N\ \delta k.\, \lambda x.\, M' \llcorner N'\ x \\
&\overset{\beta\eta}{=} N\ \delta k.\, M' \llcorner N'
\end{aligned}
$$

[DID]: Therefore $M = x$. Hence

$$
\begin{aligned}
\overline{\mathcal{D}(\!|x|\!)} \llcorner N &= \bar{x} \llcorner N \\
&= (\delta k.\, k \ulcorner x \urcorner) \llcorner N \\
&\overset{\beta\eta}{=} N \ulcorner x \urcorner \\
&= N\ x
\end{aligned}
$$

[DABS]: Therefore $M = \delta k.\, M'$ and $\Gamma\ ;\ k : K \models_{\mathrm{s}} M' : K$. Hence

$$
\begin{aligned}
\overline{\mathcal{D}(\!|\delta k.\, M'|\!)} \llcorner N &= \overline{\mathcal{K}(\!|M'|\!)} \llcorner N \\
&\overset{\beta\eta}{=} N\ \delta k.\, M' \qquad \text{by the induction hypothesis}
\end{aligned}
$$

[DFOLD]: Therefore $\Gamma\ ;\ - \models_{\mathrm{s}} M : K \multimap K$. Hence

$$
\overline{\mathcal{D}(\!|M|\!)} \llcorner N \overset{\beta\eta}{=} N\ M \quad \text{by the induction hypothesis}
$$

[DUNFOLD]: Therefore $\Gamma\ ;\ -\ \models_{\mathrm{s}} M : D$. Hence

$$\overline{\mathcal{D}(\!|M|\!)}\llcorner N \stackrel{\beta\eta}{=} N\,M \quad \text{by the induction hypothesis}$$

[PABS]: Therefore $M = \delta k.\, M'$ and $\Gamma\ ;\ k : K \models_{\mathrm{s}} M' : \mathbf{R}$. Hence

$$\begin{aligned}
\overline{\mathcal{P}(\!|\delta k.\, M'|\!)} &= \overline{\mathcal{R}(\!|M'|\!)} \\
&\stackrel{\beta\eta}{=} \delta k.\, \overline{\mathcal{R}(\!|M'|\!)}\llcorner k \\
&\stackrel{\beta\eta}{=} \delta k.\, (\delta k.\, M')\llcorner k \quad \text{by the induction hypothesis} \\
&\stackrel{\beta\eta}{=} \delta k.\, M'
\end{aligned}$$

[RAPPK]: Therefore $M = M'\,N'$, $\Gamma\ ;\ k : K \models_{\mathrm{s}} M' : K$, and $\Gamma\ ;\ -\ \models_{\mathrm{s}} N' : D$. Hence

$$\begin{aligned}
\overline{\mathcal{R}(\!|M'\,N'|\!)} &= \overline{\mathcal{K}(\!|M'|\!)\,\mathcal{D}(\!|N'|\!)} \\
&= \delta k.\, \overline{\mathcal{K}(\!|M'|\!)}\llcorner \lambda m.\, \overline{\mathcal{D}(\!|N'|\!)}\llcorner \lambda n.\, m\llcorner k\,n \\
&\stackrel{\beta\eta}{=} \delta k.\, \overline{\mathcal{K}(\!|M'|\!)}\llcorner \lambda m.\, \overline{\mathcal{D}(\!|N'|\!)}\llcorner (m\llcorner k) \\
&\stackrel{\beta\eta}{=} \delta k.\, (\lambda m.\, \overline{\mathcal{D}(\!|N'|\!)}\llcorner (m\llcorner k))\ \delta k.\, M' \qquad \text{by the induction hypothesis} \\
&\stackrel{\beta\eta}{=} \delta k.\, \overline{\mathcal{D}(\!|N'|\!)}\llcorner ((\delta k.\, M')\llcorner k) \\
&\stackrel{\beta\eta}{=} \delta k.\, \overline{\mathcal{D}(\!|N'|\!)}\llcorner M' \\
&\stackrel{\beta\eta}{=} \delta k.\, M'\,N' \qquad \text{by the induction hypothesis}
\end{aligned}$$

[RAPPP]: Therefore $M = M'\llcorner N'$, $\Gamma\ ;\ -\ \models_{\mathrm{s}} M' : K \multimap \mathbf{R}$, and $\Gamma\ ;\ k : K \models_{\mathrm{s}} N' : K$. Hence

$$\begin{aligned}
\overline{\mathcal{R}(\!|M'\llcorner N'|\!)} &= \overline{\mathcal{K}(\!|N'|\!)\,\mathcal{P}(\!|M'|\!)} \\
&= \delta k.\, \overline{\mathcal{K}(\!|N'|\!)}\llcorner \lambda m.\, \overline{\mathcal{P}(\!|M'|\!)}\llcorner \lambda n.\, m\llcorner k\,n \\
&\stackrel{\beta\eta}{=} \delta k.\, \overline{\mathcal{K}(\!|N'|\!)}\llcorner \lambda m.\, \overline{\mathcal{P}(\!|M'|\!)}\llcorner (m\llcorner k) \\
&\stackrel{\beta\eta}{=} \delta k.\, (\lambda m.\, \overline{\mathcal{P}(\!|M'|\!)}\llcorner (m\llcorner k))\ \delta k.\, N' \qquad \text{by the induction hypothesis} \\
&\stackrel{\beta\eta}{=} \delta k.\, \overline{\mathcal{P}(\!|M'|\!)}\llcorner ((\delta k.\, N')\llcorner k) \\
&\stackrel{\beta\eta}{=} \delta k.\, \overline{\mathcal{P}(\!|M'|\!)}\llcorner N' \\
&\stackrel{\beta\eta}{=} \delta k.\, M'\llcorner N' \qquad \text{by the induction hypothesis}
\end{aligned}$$

□

3.7 Other CPS Transformations

We should emphasize that the preceding analysis is not dependent on the Fischer transformation. Instead of a continuation-first transformation, we could use a continuation-second transformation [Mor93, Plo75, Rey98a] without affecting the validity of the technical results, but a continuation-first transformation admits a briefer presentation in which only a single type of continuations is needed, and continuations of this type represent control contexts. With continuation-second or

uncurried transformations, there is an additional type of continuations which are not part of the control context, and hence are not used linearly.

More explicitly, in the type used by the unrefined continuation-second transformation

$$D \stackrel{\text{def}}{=} \mu D.\, D \to (D \to \mathbf{R}) \to \mathbf{R}$$

control contexts are represented by continuations of type $D \to \mathbf{R}$, just as in the continuation-first transformation, while the continuations of type $(D \to \mathbf{R}) \to \mathbf{R}$ are not part of the control context. So in the refined version

$$D \stackrel{\text{def}}{=} \mu D.\, D \to (D \to \mathbf{R}) \multimap \mathbf{R}$$

only the former type of continuations is used linearly. Similarly, in the type used by the unrefined uncurried transformation

$$D \stackrel{\text{def}}{=} \mu D.\, (D \times (D \to \mathbf{R})) \to \mathbf{R}$$

control contexts are again represented by continuations of type $D \to \mathbf{R}$ while the continuations of type $(D \times (D \to \mathbf{R})) \to \mathbf{R}$ are not part of the control context, and hence not used linearly in the refined version[4]

$$D \stackrel{\text{def}}{=} \mu D.\, (!D \otimes (D \to \mathbf{R})) \multimap \mathbf{R}$$

Note that in both of these alternate interpretations, continuations which do not represent a control context taking continuation arguments which do represent a control context arise. This is because these former continuations are constructed before the control context of their code is known, and so they must accept a continuation argument. Hence, using continuation transformers in the standard interpretation is effectively a technique of delaying the construction of continuations until after the parent continuation is available: that is (ignoring the data context), the standard interpretation has dynamically created continuations which contain their parent continuations, while the alternate interpretations have statically created (by the transformation) continuations with dynamically determined parent continuations. This is an example of the flexibility between keeping control context internal to continuations or accepting it as an argument mentioned in Chapter 1.

3.8 Conclusion

Building on the previous chapter, here we have seen that adding recursion to the source language does not present any technical problems: the linear type system still admits the standard CPS transformation but with the return continuations restricted to linear usage:

$$D \stackrel{\text{def}}{=} \mu D.\quad \underbrace{(D \to \mathbf{R})}_{\substack{\text{return}\\ \text{continuation}}} \quad \multimap \quad \underbrace{D \to \mathbf{R}}_{\substack{\text{call}\\ \text{continuation}}}$$

But intuitively there are two questions:

1. Don't recursive calls result in "using" a continuation more than once?

2. Doesn't divergence result in ignoring a continuation, thereby "using" it zero times?

The answer to the first lies in the fact that recursion in this language is obtained by using untyped procedures to unwind to the fixed-point, and hence it is procedures, not continuations, which get

[4]We have not included $!(\cdot)$ and $(\cdot) \otimes (\cdot)$ types in the target language at this point, but here knowing the analogy is sufficient. In Section 10.9 we add $(\cdot) \otimes (\cdot)$.

"used" more than once. As for the second, the answer is that "used linearly" simply does *not* mean "invoked exactly once".

In this chapter we also established that the restriction to linear usage is actually effective, in that all the control behaviors in CPS which do not correspond to a λ-calculus term are ruled out by the linear interpretation. That is, the types and linear system exactly capture the range (not codomain) of the CPS transformation. This is shown by "carving out" the range of CPS based on the syntax of the type interpreting procedures, and then using a DS transformation inverse to the CPS transformation to witness the isomorphism.

Chapter 4

Exceptions

In this chapter we give an interpretation of exceptions which follows a discipline of linearly used control contexts. This offers a demonstration of the applicability of the technique to a quite expressive source language which provides much more flexible control behavior than procedures alone. Here we first see control contexts which are not simply a single continuation. We interpret these control contexts with an additive product, $(\cdot) \,\&\, (\cdot)$, which we use to express controlled sharing of continuations, and is central to much of the development in later chapters. The main conceptual point of this interpretation is how the sharing provided by $(\cdot) \,\&\, (\cdot)$ is sufficient to interpret exceptions, while the pattern of continuation usage entailed by exceptions does not violate the constraints which come with the sharing provided by $(\cdot) \,\&\, (\cdot)$. In this chapter we also prove no junk to demonstrate that linearity eliminates all extraneous control behaviors in not only the simplest case of procedures.

4.1 Source Language

In realistic presentations of exceptions, their typing properties are rather complex, and vary from language to language. So to eliminate complexity orthogonal to our concerns and study the jumping aspect of exceptions, we focus on an untyped source language with `raise` and `handle` primitives.

4.1.1 Syntax

We extend the syntax of terms from Section 3.1.1 with exception handling primitives:

$$
\begin{array}{rll}
V ::= & \cdots & \textit{values} \\
& \mid M\,\texttt{handle} & \text{install exception handler} \\
& \mid \texttt{raise} & \text{raise exception}
\end{array}
$$

Convention: The body of a `handle` term extends as far to the left as possible, so $M\,N\,\texttt{handle}$ parses to $(M\,N)\,\texttt{handle}$ rather than $M\,(N\,\texttt{handle})$.

Note that "`handle`" is syntactically ill-formed, "$M\,\texttt{handle}$" is a special form, not an application, and cannot be formulated as such.

4.1.2 Operational semantics

Intuitively, $M\,\texttt{handle}$ is a procedure which accepts a procedure, installs it as an exception handler, executes M, and then removes the previously installed exception handler. That is, exceptions raised during the execution of M will be handled by the argument exception handler. On the other hand, `raise` is a procedure which accepts an exception, aborts the computation up to the point where the nearest enclosing exception handler was installed, and then applies the exception handler to the exception. In a simply-typed setting the exception-handling primitives could be typed by the rules

$$\frac{}{\Gamma \vdash \texttt{raise} : \mathbf{E} \to T} \qquad \frac{\Gamma \vdash M : T}{\Gamma \vdash M\,\texttt{handle} : (\mathbf{E} \to T) \to T}$$

where $\mathbf{E}$ is the type of exceptions.

Before formally stating the operational semantics, which is an extension of the big-step semantics of Section 3.1.2, we illustrate with several examples.[1] If the body of a `handle` expression evaluates to a value, the handler is ignored:

$$42\,\texttt{handle}\,\lambda e.\, e + 1 \Downarrow 42$$

On the other hand, if the body raises a value, the handler is applied to it:

$$\texttt{raise}\,41\,\texttt{handle}\,\lambda e.\, e + 1 \Downarrow 42$$

When the body raises a value, any unevaluated portion of the body is ignored and the handler is applied immediately:

$$1 - (\texttt{raise}\,41)\,\texttt{handle}\,\lambda e.\, e + 1 \Downarrow 42$$
$$\texttt{raise}\,(\texttt{raise}\,41)\,\texttt{handle}\,\lambda e.\, e + 1 \Downarrow 42$$

When the body of a `handle` expression raises, the nearest enclosing handler is applied:

$$\texttt{raise}\,13\,\texttt{handle}\,\lambda e.\, e + 29\,\texttt{handle}\,\lambda e.\, e + 1 \Downarrow 42$$

When the body of a handler raises, the next enclosing handler is applied:

$$\texttt{raise}\,13\,\texttt{handle}\,\lambda e.\,\texttt{raise}\,(e + 28)\,\texttt{handle}\,\lambda e.\, e + 1 \Downarrow 42$$

Finally, a characteristic feature of exceptions: When a value is raised, it is the *dynamically* enclosing handler which is applied. The *statically* (or lexically) enclosing handler, the nearest enclosing handler in the program code, has no significance. Hence when the body of a `handle` expression evaluates to a value, the handler is forgotten and will never be applied:

$$\frac{\dfrac{}{\texttt{raise}\,\texttt{handle}\,\lambda e.\, e - 1 \Downarrow \texttt{raise}} \qquad \dfrac{}{\texttt{raise}\,41\,\texttt{handle}\,\lambda e.\, e + 1 \Downarrow 42}}{(\lambda f.\, f\,41\,\texttt{handle}\,\lambda e.\, e + 1)\,(\underline{\texttt{raise}}\,\texttt{handle}\,\underline{\lambda e.\, e - 1}) \Downarrow 42}$$

So, there is no connection between `raise` and $\lambda e.\, e - 1$ underlined above. Instead, `raise` refers to the nearest enclosing handler when a value is raised, namely $\lambda e.\, e + 1$.

Formally, as in [BK01], the big-step operational semantics is given by two mutually recursive judgments: $M \Downarrow V$ which states that term M evaluates to value V, and $M \Uparrow V$ which states that evaluation of term M raises exception value V. The rules for $M \Downarrow V$ are an extension of those in Section 3.1.2.

[1] Though we have not treated numeric constants and operations, we make use of them in the illustrations.

$$[\textsc{Handle1}]\ \frac{M_0 \Downarrow M_2\,\texttt{handle} \quad M_1 \Downarrow V_1 \quad M_2 \Downarrow V_0}{M_0\,M_1 \Downarrow V_0}$$

$$[\textsc{Handle2}]\ \frac{M_0 \Downarrow M_2\,\texttt{handle} \quad M_1 \Downarrow V_1 \quad M_2 \Uparrow V_2 \quad V_1\,V_2 \Downarrow V_0}{M_0\,M_1 \Downarrow V_0}$$

$$[\textsc{Raise}]\ \frac{M_0 \Downarrow \texttt{raise} \quad M_1 \Downarrow V}{M_0\,M_1 \Uparrow V} \qquad [\textsc{Prop1}]\ \frac{M_0 \Uparrow V}{M_0\,M_1 \Uparrow V} \qquad [\textsc{Prop2}]\ \frac{M_0 \Downarrow V_1 \quad M_1 \Uparrow V_0}{M_0\,M_1 \Uparrow V_0}$$

$$[\textsc{Prop3}]\ \frac{M_0 \Downarrow \lambda x.\,M_2 \quad M_1 \Downarrow V_1 \quad M_2[x \mapsto V_1] \Uparrow V_0}{M_0\,M_1 \Uparrow V_0}$$

$$[\textsc{Prop4}]\ \frac{M_0 \Downarrow M_2\,\texttt{handle} \quad M_1 \Downarrow V_1 \quad M_2 \Uparrow V_2 \quad V_1\,V_2 \Uparrow V_0}{M_0\,M_1 \Uparrow V_0}$$

4.2 Target Language $(+= (\cdot)\ \&\ (\cdot))$

To treat exceptions we add an additive (as opposed to multiplicative) product to the target language. The additive product essentially enable us to express constraints such as either of two resources may be used, but not both.

4.2.1 Syntax

We extend the grammar of terms of Section 2.2 with the productions

$M ::= \cdots$	*terms*	
$\mid \langle M, M\rangle$	additive pair	
$\mid \pi_i$	additive projection	$i \in \{0,1\}$

In addition, in Section A.1 we define the following syntactic sugar:

$\mid \delta\langle x_0, x_1\rangle.\,M$	restricted additive pattern match

Convention: $\&$-pairing is right-associative, so $\langle M, N, O\rangle$ parses to $\langle M, \langle N, O\rangle\rangle$ rather than $\langle\langle M, N\rangle, O\rangle$. Pattern match terms parse like abstractions.

4.2.2 Equational theory

Additive pairs, $\langle M, M\rangle$, and projections, π_i, differ from the standard multiplicative versions in typing, but operationally and equationally they are equivalent. So we add the following to the axioms of Section 3.2:

$$\frac{}{\pi_i \llcorner \langle M_0, M_1\rangle \overset{\beta\eta}{=} M_i} \qquad \frac{}{\langle \pi_0 \llcorner M, \pi_1 \llcorner M\rangle \overset{\beta\eta}{=} M}$$

And, in Section A.2 we define the following axiom for the syntactic sugar:

$$\frac{}{(\delta\langle x_0, x_1\rangle.\,M) \llcorner \langle M_0, M_1\rangle \overset{\beta\eta}{=} M[x_0, x_1 \mapsto M_0, M_1]}$$

4.2.3 Type system

The grammar of types of Section 3.2 is extended by

$$\begin{array}{rll} P ::= & \cdots & \textit{pointed types} \\ & \mid P \mathbin{\&} P & \text{additive product type} \end{array}$$

Convention: The additive product type constructor is right-associative, so $P \mathbin{\&} Q \mathbin{\&} R$ parses to $P \mathbin{\&} (Q \mathbin{\&} R)$ rather than $(P \mathbin{\&} Q) \mathbin{\&} R$. The precedence of $(\cdot) \mathbin{\&} (\cdot)$ is higher than $(\cdot) \to (\cdot)$ and $(\cdot) \multimap (\cdot)$, so $P \mathbin{\&} Q \multimap R$ parses to $(P \mathbin{\&} Q) \multimap R$ rather than $P \mathbin{\&} (Q \multimap R)$.

The typing rules of Section 3.2 are extended with

$$[\textsc{Apair}]\ \frac{\Gamma\ ;\ \Delta \vdash M : P \quad \Gamma\ ;\ \Delta \vdash N : Q}{\Gamma\ ;\ \Delta \vdash \langle M, N \rangle : P \mathbin{\&} Q} \qquad [\textsc{Aproj}i]\ \frac{}{\Gamma\ ;\ - \vdash \pi_i : P_0 \mathbin{\&} P_1 \multimap P_i}$$

And, in Section A.3 we define judgments with $\&$-pairs in the context for the syntactic sugar, and derive the following axiom:

$$[\textsc{Rapid}]\ \frac{}{\Gamma\ ;\ \langle x_0, x_1 \rangle : P_0 \mathbin{\&} P_1 \vdash x_i : P_i}$$

These rules ensure that both factors in a $\&$-pair depend on the same linear identifiers, but when one factor is projected from the $\&$-pair, there is no way to access the other. So individual factors of a $\&$-pair may be duplicated and discarded within a $\&$-pair, but the $\&$-pair itself may not. For instance, we have

$$-\ ;\ x : P \mathbin{\&} Q \vdash \langle \pi_0 \mathbin{\lrcorner} x, \pi_0 \mathbin{\lrcorner} x \rangle : P \mathbin{\&} P$$

in which the left factor is duplicated and the right is discarded within a $\&$-pair, but neither

$$-\ ;\ x : (P \multimap Q) \mathbin{\&} P \vdash \pi_0 \mathbin{\lrcorner} x \mathbin{\lrcorner} (\pi_1 \mathbin{\lrcorner} x) : Q$$

nor

$$y : A\ ;\ x : P \mathbin{\&} Q \vdash y : A$$

is derivable since a $\&$-pair is duplicated in the first judgment and discarded in the second.

When a $\&$-pair is the argument to an intuitionistic function it can be duplicated and discarded freely, in which case $(\cdot) \mathbin{\&} (\cdot)$ is simply the usual Cartesian product.

4.3 Refined CPS Transformation

As the examples in Section 4.1.2 show, evaluating an expression will result in either returning or raising a value. Hence the control context can be represented as two continuations: return and handler. Just as in the interpretation of λ-calculus, the return continuation represents the return address, and so returning from a procedure is interpreted by invoking the return continuation. Likewise, the current exception handler is represented as the handler continuation, and so raising an exception is interpreted by invoking the handler continuation with the exception. Since an expression cannot both return and raise, only one of the return and handler continuations will be invoked. And since an expression must either return or raise, one of the return and handler continuations must be invoked. So if we take the control context to be a $\&$-pair of the return and

handler continuations, the typing behavior of additive products precisely captures the possible operational behavior. Conceptually, a &-pair of continuations is two continuations which may share some internal control environment. Linear use of these &-pairs means that (one and) only one continuation may (and must) be used, so in particular, invoking one continuation passing the other as the argument is disallowed. In later chapters we will see that this idea generalizes to &-tuples, and the interpretation of λ-calculus can be seen as simply a degenerate case.

Formally, we interpret source procedures with the type

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to \mathbf{R})}_{\substack{\text{return}\\ \text{continuation}}}\ \&\ \underbrace{(D \to \mathbf{R})}_{\substack{\text{handler}\\ \text{continuation}}}\ \multimap\ \underbrace{D \to \mathbf{R}}_{\substack{\text{call}\\ \text{continuation}}}$$

which we somewhat abusively refer to as a type of *continuation transformers* since it maps multiple continuations to another continuation. We sometimes use the abbreviation

$$T \stackrel{\text{def}}{=} (D \to \mathbf{R}) \,\&\, (D \to \mathbf{R})$$

A version of this interpretation for a simply-typed source language can be derived from a direct semantics, following [Mog91]. That is, we start with

$$(A \to B)^* = A^* \to B^* + \mathbf{E}$$

followed by a standard continuation semantics, which gives us

$$\overline{A^* \to B^* + \mathbf{E}} = (\overline{B^*} + \overline{\mathbf{E}} \to \mathbf{R}) \multimap \overline{A^*} \to \mathbf{R}$$

and finally, a manipulation using the isomorphism

$$\overline{B^*} + \overline{\mathbf{E}} \to \mathbf{R} \cong (\overline{B^*} \to \mathbf{R}) \,\&\, (\overline{\mathbf{E}} \to \mathbf{R})$$

yields

$$\overline{A^* \to B^* + \mathbf{E}} = (\overline{B^*} \to \mathbf{R}) \,\&\, (\overline{\mathbf{E}} \to \mathbf{R}) \multimap \overline{A^*} \to \mathbf{R}$$

The double-barreled [KYD98, Thi02] transformations of terms and of values, given in Figure 4.1, are mutually recursive. The transformation of `raise`

$$\ulcorner \texttt{raise} \urcorner \stackrel{\text{def}}{=} \delta\langle r, h\rangle.\, \lambda x.\, h\, x$$

accepts return and handler continuations, then an argument, and then invokes the supplied handler continuation with the argument, discarding (the unshared part of) the return continuation. Similarly, the transformation of a value

$$\overline{V} \stackrel{\text{def}}{=} \delta\langle r, h\rangle.\, r\, \ulcorner V \urcorner$$

discards the handler continuation. On the other hand, the transformation of M `handle`

$$\ulcorner M\ \texttt{handle} \urcorner \stackrel{\text{def}}{=} \delta\langle r, h\rangle.\, \lambda x.\, \overline{M}_{\sqcup}\langle r, \lambda m.\, x_{\sqcup}\langle r, h\rangle\, m\rangle \quad x \notin \mathsf{fi}(M)$$

accepts return and handler continuations, then an argument exception handler procedure, and then evaluates the body with the same return continuation but installs a new handler continuation obtained from applying the handler procedure to the return and handler continuations, sharing the return continuation. Similarly, the transformation of an application

$$\overline{M\,N} \stackrel{\text{def}}{=} \delta\langle r,h\rangle.\,\overline{M}_{\sqcup}\langle\lambda m.\,\overline{N}_{\sqcup}\langle\lambda n.\,m_{\sqcup}\langle r,h\rangle\,n,h\rangle,h\rangle \quad m \notin \mathsf{fi}(N)$$

shares the handler continuation. The following equalities better explicate the symmetries between returning and raising and between application and exception handler installation:

$$\begin{aligned}
\overline{V} &\stackrel{\text{def}}{=} \delta\langle r,h\rangle.\,r\,\ulcorner V\urcorner \\
\overline{\texttt{raise}\,V} &\stackrel{\beta\eta}{=} \delta\langle r,h\rangle.\,h\,\ulcorner V\urcorner \\
\overline{M\,N} &\stackrel{\text{def}}{=} \delta\langle r,h\rangle.\,\overline{M}_{\sqcup}\langle\lambda m.\,\overline{N}_{\sqcup}\langle\lambda n.\,m_{\sqcup}\langle r,h\rangle\,n,h\rangle,h\rangle \quad m \notin \mathsf{fi}(N) \\
\overline{M\,\texttt{handle}\,N} &\stackrel{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}_{\sqcup}\langle\lambda n.\,\overline{M}_{\sqcup}\langle r,\lambda m.\,n_{\sqcup}\langle r,h\rangle\,m\rangle,h\rangle \quad n \notin \mathsf{fi}(M)
\end{aligned}$$

We try to give a feel for the jumpy flavor of this semantics with the following example:

$$\overline{x\,(\texttt{raise}\,y)\,\texttt{handle}\,\lambda e.\,H} \stackrel{\beta\eta}{=} \delta\langle r,h\rangle.\,(\lambda m.\,(\lambda e.\,\overline{H}_{\sqcup}\langle r,h\rangle)\,y)\,x \quad m \notin \mathsf{fi}(\lambda e.\,H)$$

Here x and y are free identifiers which an environment will give values to. The CPS version, when given return and handler continuations, throws away the value of x, binds e to the value of y, and then runs the body of the handler procedure in the control context of the original term. Notice that the value of x is never applied to anything, as would be the case if `raise` y was interpreted as returning some special value which a cooked application knew to pass upward. Instead the interpretation of `raise` y jumps past the remaining code directly to the handler. So while this semantics is in some sense equivalent to the $+\,\mathbf{E}$ semantics mentioned earlier, in another sense it is very different.

The interpretation of programs using the type

$$\underbrace{(D \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{R}$$

is slightly less trivial than the others we have presented, see Figure 4.1. Here we have, mostly arbitrarily, chosen to pass unhandled exceptions to the toplevel continuation.

Figure 4.1 Refined CPS of Exceptions

Value Terms

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to \mathbf{R})}_{\text{return}} \,\&\, \underbrace{(D \to \mathbf{R})}_{\text{handler}} \multimap \underbrace{D \to \mathbf{R}}_{\text{call}}$$

$$\begin{aligned}
\ulcorner x \urcorner &\stackrel{\text{def}}{=} x \\
\ulcorner \lambda x.\, M \urcorner &\stackrel{\text{def}}{=} \delta\langle r, h\rangle.\, \lambda x.\, \overline{M}\langle r, h\rangle \\
\ulcorner \texttt{raise} \urcorner &\stackrel{\text{def}}{=} \delta\langle r, h\rangle.\, \lambda x.\, h\, x \\
\ulcorner M\, \texttt{handle} \urcorner &\stackrel{\text{def}}{=} \delta\langle r, h\rangle.\, \lambda x.\, \overline{M}\langle r, \lambda m.\, x\langle r, h\rangle\, m\rangle \quad x \notin \mathsf{fi}(M)
\end{aligned}$$

Terms

$$\underbrace{(D \to \mathbf{R})}_{\text{return}} \,\&\, \underbrace{(D \to \mathbf{R})}_{\text{handler}} \multimap \mathbf{R}$$

$$\begin{aligned}
\overline{V} &\stackrel{\text{def}}{=} \delta\langle r, h\rangle.\, r\, \ulcorner V \urcorner \\
\overline{M\, N} &\stackrel{\text{def}}{=} \delta\langle r, h\rangle.\, \overline{M}\langle \lambda m.\, \overline{N}\langle \lambda n.\, m\langle r, h\rangle\, n, h\rangle, h\rangle \quad m \notin \mathsf{fi}(N)
\end{aligned}$$

Programs

$$\underbrace{(D \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{R}$$

$$\llcorner M \lrcorner \stackrel{\text{def}}{=} \delta k.\, \overline{M}\langle k, k\rangle$$

4.4 Soundness

The soundness statement and proof are very similar to those in Chapter 3.

Proposition 8 (Soundness) *For any source term M, if $\{x_1, \ldots, x_n\} \supseteq \mathsf{fi}(M)$ for some $n \geq 0$, then*

1. $$x_1 : D\,, \ldots, x_n : D\ ;\ - \vdash \overline{M} : (D \to \mathbf{R}) \,\&\, (D \to \mathbf{R}) \multimap \mathbf{R}$$

   and if M is a value, then

   $$x_1 : D\,, \ldots, x_n : D\ ;\ - \vdash \ulcorner M \urcorner : D$$

2. $$x_1 : D\,, \ldots, x_n : D\ ;\ - \vdash \llcorner M \lrcorner : (D \to \mathbf{R}) \multimap \mathbf{R}$$

Proof Let $\Gamma = x_1 : D\,, \ldots, x_n : D$.

1. Proceed by structural induction on the syntax of M:

   $[V]$: Therefore the induction hypothesis ensures

   $$\Gamma\ ;\ - \vdash \ulcorner V \urcorner : D$$

from which the target derivation follows:

$$\dfrac{\dfrac{\overline{\Gamma\ ;\ \langle r,h\rangle : T \vdash r : \neg D} \qquad \Gamma\ ;\ - \vdash \ulcorner V \urcorner : D}{\Gamma\ ;\ \langle r,h\rangle : T \vdash r\ \ulcorner V \urcorner : \mathbf{R}}}{\Gamma\ ;\ - \vdash \delta\langle r,h\rangle .\, r\ \ulcorner V \urcorner : T \multimap \mathbf{R}}$$

$[N\,O]$: Therefore the induction hypothesis ensures

$$\Gamma\ ;\ - \vdash \overline{N} : T \multimap \mathbf{R}$$

and

$$\Gamma\ ;\ - \vdash \overline{O} : T \multimap \mathbf{R}$$

Let $\Gamma' = \Gamma\ , m : D$, $\Gamma'' = \Gamma'\ , n : D$ and we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma''\ ;\ - \vdash m : D}}{\Gamma''\ ;\ - \vdash m : T \multimap \neg D} \qquad \overline{\Gamma''\ ;\ \langle r,h\rangle : T \vdash \langle r,h\rangle : T}}{\Gamma''\ ;\ \langle r,h\rangle : T \vdash m\llcorner\langle r,h\rangle : \neg D} \qquad \overline{\Gamma''\ ;\ - \vdash n : D}}{\Gamma''\ ;\ \langle r,h\rangle : T \vdash m\llcorner\langle r,h\rangle\ n : \mathbf{R}}}{\Gamma'\ ;\ \langle r,h\rangle : T \vdash \lambda n.\, m\llcorner\langle r,h\rangle\ n : \neg D} \qquad \overline{\Gamma'\ ;\ \langle r,h\rangle : T \vdash h : \neg D}}{\dfrac{\Gamma'\ ;\ - \vdash \overline{O} : T \multimap \mathbf{R} \qquad \Gamma'\ ;\ \langle r,h\rangle : T \vdash \langle \lambda n.\, m\llcorner\langle r,h\rangle\ n, h\rangle : T}{\Gamma'\ ;\ \langle r,h\rangle : T \vdash \overline{O}\llcorner\langle \lambda n.\, m\llcorner\langle r,h\rangle\ n, h\rangle : \mathbf{R}}}}{\Gamma\ ;\ \langle r,h\rangle : T \vdash \lambda m.\, \overline{O}\llcorner\langle \lambda n.\, m\llcorner\langle r,h\rangle\ n, h\rangle : \neg D} \qquad \overline{\Gamma\ ;\ \langle r,h\rangle : T \vdash h : \neg D}}{\dfrac{\Gamma\ ;\ - \vdash \overline{N} : T \multimap \mathbf{R} \qquad \Gamma\ ;\ \langle r,h\rangle : T \vdash \langle \lambda m.\, \overline{O}\llcorner\langle \lambda n.\, m\llcorner\langle r,h\rangle\ n, h\rangle, h\rangle : T}{\Gamma\ ;\ \langle r,h\rangle : T \vdash \overline{N}\llcorner\langle \lambda m.\, \overline{O}\llcorner\langle \lambda n.\, m\llcorner\langle r,h\rangle\ n, h\rangle, h\rangle : \mathbf{R}}}{\Gamma\ ;\ - \vdash \delta\langle r,h\rangle .\, \overline{N}\llcorner\langle \lambda m.\, \overline{O}\llcorner\langle \lambda n.\, m\llcorner\langle r,h\rangle\ n, h\rangle, h\rangle : T \multimap \mathbf{R}}$$

Now suppose $M = V$ for some value V and proceed by cases on the syntax of V:

$[x]$: $x = x_i$ for $1 \leq i \leq n$ and the target derivation is immediate:

$$\overline{\Gamma\ ;\ - \vdash x : D}$$

$[\lambda x.\, N]$: Therefore the induction hypothesis ensures

$$\Gamma\ , x : D\ ;\ - \vdash \overline{N} : T \multimap \mathbf{R}$$

from which the target derivation follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma\ , x : D\ ;\ - \vdash \overline{N} : T \multimap \mathbf{R} \qquad \overline{\Gamma\ , x : D\ ;\ \langle r,h\rangle : T \vdash \langle r,h\rangle : T}}{\Gamma\ , x : D\ ;\ \langle r,h\rangle : T \vdash \overline{N}\llcorner\langle r,h\rangle : \mathbf{R}}}{\Gamma\ ;\ \langle r,h\rangle : T \vdash \lambda x.\, \overline{N}\llcorner\langle r,h\rangle : \neg D}}{\Gamma\ ;\ - \vdash \delta\langle r,h\rangle .\, \lambda x.\, \overline{N}\llcorner\langle r,h\rangle : T \multimap \neg D}}{\Gamma\ ;\ - \vdash \delta\langle r,h\rangle .\, \lambda x.\, \overline{N}\llcorner\langle r,h\rangle : D}$$

[N `handle`]: Therefore the induction hypothesis ensures

$$\Gamma\ ;\ -\ \vdash \overline{N} : T \multimap \mathbf{R}$$

Let $\Gamma' = \Gamma\ , x : D$, $\Gamma'' = \Gamma'\ , m : D$ and we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma''\ ;\ -\ \vdash x : D}}{\Gamma''\ ;\ -\ \vdash x : T \multimap \neg D} \quad \dfrac{}{\Gamma''\ ;\ \langle r,h\rangle : T \vdash \langle r,h\rangle : T}}{\Gamma''\ ;\ \langle r,h\rangle : T \vdash x\,\_\langle r,h\rangle : \neg D} \quad \dfrac{}{\Gamma''\ ;\ -\ \vdash m : D}}{\Gamma''\ ;\ \langle r,h\rangle : T \vdash x\,\_\langle r,h\rangle\ m : \mathbf{R}}}{\Gamma'\ ;\ \langle r,h\rangle : T \vdash \lambda m.\, x\,\_\langle r,h\rangle\ m : \neg D} \quad \dfrac{}{\Gamma'\ ;\ \langle r,h\rangle : T \vdash r : \neg D}}{\Gamma'\ ;\ \langle r,h\rangle : T \vdash \langle r, \lambda m.\, x\,\_\langle r,h\rangle\ m\rangle : T} \quad \dfrac{}{\Gamma'\ ;\ -\ \vdash \overline{N} : T \multimap \mathbf{R}}}{\Gamma'\ ;\ \langle r,h\rangle : T \vdash \overline{N}\,\_\langle r, \lambda m.\, x\,\_\langle r,h\rangle\ m\rangle : \mathbf{R}}}{\Gamma\ ;\ \langle r,h\rangle : T \vdash \lambda x.\, \overline{N}\,\_\langle r, \lambda m.\, x\,\_\langle r,h\rangle\ m\rangle : \neg D}}{\Gamma\ ;\ -\ \vdash \delta\langle r,h\rangle.\, \lambda x.\, \overline{N}\,\_\langle r, \lambda m.\, x\,\_\langle r,h\rangle\ m\rangle : T \multimap \neg D}}{\Gamma\ ;\ -\ \vdash \delta\langle r,h\rangle.\, \lambda x.\, \overline{N}\,\_\langle r, \lambda m.\, x\,\_\langle r,h\rangle\ m\rangle : D}\ T \multimap \neg D = D$$

[`raise`]: The target derivation is immediate:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\ , x : D\ ;\ \langle r,h\rangle : T \vdash h : \neg D} \quad \dfrac{}{\Gamma\ , x : D\ ;\ -\ \vdash x : D}}{\Gamma\ , x : D\ ;\ \langle r,h\rangle : T \vdash h\,x : \mathbf{R}}}{\Gamma\ ;\ \langle r,h\rangle : T \vdash \lambda x.\, h\,x : \neg D}}{\Gamma\ ;\ -\ \vdash \delta\langle r,h\rangle.\, \lambda x.\, h\,x : T \multimap \neg D}}{\Gamma\ ;\ -\ \vdash \delta\langle r,h\rangle.\, \lambda x.\, h\,x : D}$$

2. By 1 we have

$$\Gamma\ ;\ -\ \vdash \overline{M} : (D \to \mathbf{R})\ \&\ (D \to \mathbf{R}) \multimap \mathbf{R}$$

from which the target derivation follows

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\ ;\ -\ \vdash \overline{M} : \neg D\ \&\ \neg D \multimap \mathbf{R}} \quad \dfrac{\dfrac{}{\Gamma\ ;\ k : \neg D \vdash k : \neg D} \quad \dfrac{}{\Gamma\ ;\ k : \neg D \vdash k : \neg D}}{\Gamma\ ;\ k : \neg D \vdash \langle k,k\rangle : \neg D\ \&\ \neg D}}{\Gamma\ ;\ k : \neg D \vdash \overline{M}\,\_\langle k,k\rangle : \mathbf{R}}}{\Gamma\ ;\ -\ \vdash \delta k.\, \overline{M}\,\_\langle k,k\rangle : \neg D \multimap \mathbf{R}}$$

□

4.5 Completeness

The completeness result and development for the language of exceptions is very similar to that for λ-calculus and so we tersely present the technicalities.

Proposition 9 (No Junk) *If* $\Gamma\ ;\ -\ \models_{\mathrm{s}} M : (D \to \mathbf{R})\ \&\ (D \to \mathbf{R}) \multimap \mathbf{R}$*, then there exists a source term N such that* $M \overset{\beta\eta}{=} \overline{N}$.

4.5.1 Target sublanguage

As with procedures, we define a sublanguage of the general target language by restricting the form of types to only those used by the CPS transformation.

The syntax of terms and the equational theory are unchanged from Section 4.2. Since the interpretation in Section 4.3 revolves around the type

$$D = \mu D.\, (D \to \mathbf{R}) \,\&\, (D \to \mathbf{R}) \multimap D \to \mathbf{R}$$

the syntax of types is built from it:

$$\begin{array}{lll}
A, P ::= & & \textit{types} \\
\quad | & D \to \mathbf{R} & \text{continuation type} \\
\quad | & (D \to \mathbf{R}) \,\&\, (D \to \mathbf{R}) & \text{continuation \&-pair type} \\
\quad | & D \mid (D \to \mathbf{R}) \,\&\, (D \to \mathbf{R}) \multimap D \to \mathbf{R} & \text{continuation transformer types} \\
\quad | & (D \to \mathbf{R}) \,\&\, (D \to \mathbf{R}) \multimap \mathbf{R} & \text{program type} \\
\quad | & \mathbf{R} & \text{result type}
\end{array} \tag{4.1}$$

The typing rules of Section 4.2 specialized to the restricted form of types are

$$\frac{\Gamma\,,\, x : D\,;\, \Delta \models_{\mathrm{s}} M : \mathbf{R}}{\Gamma\,;\, \Delta \models_{\mathrm{s}} \lambda x.\, M : \neg D} \qquad \frac{\Gamma\,;\, \Delta \models_{\mathrm{s}} M : T \multimap \neg D \quad \Gamma\,;\, \Delta' \models_{\mathrm{s}} N : T}{\Gamma\,;\, \Delta\,,\, \Delta' \models_{\mathrm{s}} M \llcorner N : \neg D}$$

$$\frac{}{\Gamma\,;\, t : T \models_{\mathrm{s}} t : T} \qquad \frac{\Gamma\,;\, \Delta \models_{\mathrm{s}} M : \neg D \quad \Gamma\,;\, \Delta \models_{\mathrm{s}} N : \neg D}{\Gamma\,;\, \Delta \models_{\mathrm{s}} \langle M, N \rangle : T}$$

$$\frac{}{\Gamma\,,\, x : D\,;\, - \models_{\mathrm{s}} x : D} \qquad \frac{\Gamma\,;\, \Delta\,,\, t : T \models_{\mathrm{s}} M : \neg D}{\Gamma\,;\, \Delta \models_{\mathrm{s}} \delta t.\, M : T \multimap \neg D} \qquad \frac{}{\Gamma\,;\, - \models_{\mathrm{s}} \pi_i : T \multimap \neg D}$$

$$\frac{\Gamma\,;\, \Delta \models_{\mathrm{s}} M : T \multimap \neg D}{\Gamma\,;\, \Delta \models_{\mathrm{s}} M : D} \qquad \frac{\Gamma\,;\, \Delta \models_{\mathrm{s}} M : D}{\Gamma\,;\, \Delta \models_{\mathrm{s}} M : T \multimap \neg D}$$

$$\frac{\Gamma\,;\, \Delta\,,\, t : T \models_{\mathrm{s}} M : \mathbf{R}}{\Gamma\,;\, \Delta \models_{\mathrm{s}} \delta t.\, M : T \multimap \mathbf{R}}$$

$$\frac{\Gamma\,;\, \Delta \models_{\mathrm{s}} M : \neg D \quad \Gamma\,;\, - \models_{\mathrm{s}} N : D}{\Gamma\,;\, \Delta \models_{\mathrm{s}} M N : \mathbf{R}} \qquad \frac{\Gamma\,;\, \Delta \models_{\mathrm{s}} M : T \multimap \mathbf{R} \quad \Gamma\,;\, \Delta' \models_{\mathrm{s}} N : T}{\Gamma\,;\, \Delta\,,\, \Delta' \models_{\mathrm{s}} M \llcorner N : \mathbf{R}}$$

Observation 10 $\Gamma\,;\, \Delta \models_{\mathrm{s}} M : A$ *if and only if* $\Gamma\,;\, \Delta \vdash M : A$ *and all types which appear in the derivation of the latter judgment are in the type language defined by (4.1).*

Note that all the derivations we have presented so far satisfy this property.

Observation 11 *If* $\Gamma\,;\, \Delta \models_{\mathrm{s}} M : A$*, then* $\Delta = -$ *or* $\Delta = t : T$*.*

Taking this special form of contexts into account, we can further specialize the presentation of the typing rules:

$$[\text{KABS}]\ \frac{\Gamma, x:D\ ;\ t:T \models_{\text{s}} M:\mathbf{R}}{\Gamma\ ;\ t:T \models_{\text{s}} \lambda x.M:\neg D} \qquad [\text{KAPP}]\ \frac{\Gamma\ ;\ -\models_{\text{s}} M:T \multimap \neg D \quad \Gamma\ ;\ t:T \models_{\text{s}} N:T}{\Gamma\ ;\ t:T \models_{\text{s}} M \llcorner N:\neg D}$$

$$[\text{TID}]\ \frac{}{\Gamma\ ;\ t:T \models_{\text{s}} t:T} \qquad [\text{TPAIR}]\ \frac{\Gamma\ ;\ t:T \models_{\text{s}} M:\neg D \quad \Gamma\ ;\ t:T \models_{\text{s}} N:\neg D}{\Gamma\ ;\ t:T \models_{\text{s}} \langle M,N\rangle:T}$$

$$[\text{DID}]\ \frac{}{\Gamma, x:D\ ;\ -\models_{\text{s}} x:D} \quad [\text{DABS}]\ \frac{\Gamma\ ;\ t:T \models_{\text{s}} M:\neg D}{\Gamma\ ;\ -\models_{\text{s}} \delta t.M:T \multimap \neg D} \quad [\text{DPROJ}i]\ \frac{}{\Gamma\ ;\ -\models_{\text{s}} \pi_i:T \multimap \neg D}$$

$$[\text{DFOLD}]\ \frac{\Gamma\ ;\ -\models_{\text{s}} M:T \multimap \neg D}{\Gamma\ ;\ -\models_{\text{s}} M:D} \qquad [\text{DUNFOLD}]\ \frac{\Gamma\ ;\ -\models_{\text{s}} M:D}{\Gamma\ ;\ -\models_{\text{s}} M:T \multimap \neg D}$$

$$[\text{PABS}]\ \frac{\Gamma\ ;\ t:T \models_{\text{s}} M:\mathbf{R}}{\Gamma\ ;\ -\models_{\text{s}} \delta t.M:T \multimap \mathbf{R}}$$

$$[\text{RAPPK}]\ \frac{\Gamma\ ;\ t:T \models_{\text{s}} M:\neg D \quad \Gamma\ ;\ -\models_{\text{s}} N:D}{\Gamma\ ;\ t:T \models_{\text{s}} MN:\mathbf{R}}$$

$$[\text{RAPPP}]\ \frac{\Gamma\ ;\ -\models_{\text{s}} M:T \multimap \mathbf{R} \quad \Gamma\ ;\ t:T \models_{\text{s}} N:T}{\Gamma\ ;\ t:T \models_{\text{s}} M \llcorner N:\mathbf{R}}$$

4.5.2 DS transformation

The DS transformation of programs $\mathcal{P}(\!|(\cdot)|\!)$ is defined mutually recursively with $\mathcal{K}(\!|(\cdot)|\!)$, $\mathcal{T}(\!|(\cdot)|\!)$, $\mathcal{D}(\!|(\cdot)|\!)$, and $\mathcal{R}(\!|(\cdot)|\!)$, which transform continuations, continuation &-pairs, continuation transformers, and results, respectively:

$$\begin{aligned}
\mathcal{K}(\!|\lambda x.M|\!) &\stackrel{\text{def}}{=} \lambda x.\,\mathcal{R}(\!|M|\!) \\
\mathcal{K}(\!|M \llcorner N|\!) &\stackrel{\text{def}}{=} \lambda x.\,\mathcal{T}(\!|N|\!)[\mathcal{D}(\!|M|\!)\,x] && x \notin \mathsf{fi}(M \llcorner N)
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}(\!|t|\!) &\stackrel{\text{def}}{=} [\,] \\
\mathcal{T}(\!|\langle M,N\rangle|\!) &\stackrel{\text{def}}{=} (\lambda y.\,\lambda x.\,x\,y)\,[\,]\,\texttt{handle}\,(\lambda e.\,\lambda x.\,\mathcal{K}(\!|N|\!)\,e)\,\mathcal{K}(\!|M|\!) && x \notin \mathsf{fi}(N)
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}(\!|x|\!) &\stackrel{\text{def}}{=} x \\
\mathcal{D}(\!|\delta t.M|\!) &\stackrel{\text{def}}{=} \mathcal{K}(\!|M|\!) \\
\mathcal{D}(\!|\pi_0|\!) &\stackrel{\text{def}}{=} \lambda x.\,x \\
\mathcal{D}(\!|\pi_1|\!) &\stackrel{\text{def}}{=} \texttt{raise}
\end{aligned}$$

$$\mathcal{P}(\!|\delta t.M|\!) \stackrel{\text{def}}{=} \mathcal{R}(\!|M|\!)$$

$$\begin{aligned}
\mathcal{R}(\!|M\,N|\!) &\stackrel{\text{def}}{=} \mathcal{K}(\!|M|\!)\,\mathcal{D}(\!|N|\!) \\
\mathcal{R}(\!|M \llcorner N|\!) &\stackrel{\text{def}}{=} \mathcal{T}(\!|N|\!)[\mathcal{P}(\!|M|\!)]
\end{aligned}$$

Where $[\,]$ is a "hole" in a term and we write $M[N]$ for the result of filling the hole in M with N. Note that this is not equivalent to abstraction and application since we fill the hole with an application in the clause for $\mathcal{K}(\!|M \llcorner N|\!)$ and the full β law is not valid in the source language, only β-value is valid.

The key of this DS transformation, like the one in Section 3.6.3, is the transformation of control contexts: here continuation &-pairs instead of continuations. The idea here is also to

replace explicit manipulation of control contexts by introduction of implicit control contexts, but now the control contexts are more complicated. For instance

$$\mathcal{K}(\!|(\delta t.\, M)\llcorner\langle R, H\rangle|\!) = \lambda z.\, (\lambda y.\, \lambda x.\, x\, y)\, (\mathcal{K}(\!|M|\!)\, z)\, \texttt{handle}\, (\lambda e.\, \lambda x.\, \mathcal{K}(\!|H|\!)\, e)\, \mathcal{K}(\!|R|\!)$$

puts the DS transformation of continuation M in the context of not only an application (eventually) to the transformation of its return continuation, but also an exception handler built from the DS transformation of the handler continuation. Care must be taken to ensure that exceptions raised when $\mathcal{K}(\!|R|\!)$ is called are not handled by the newly installed handler, which would not be properly dealt with by

$$\lambda z.\, (\lambda y.\, \mathcal{K}(\!|R|\!)\, y)\, (\mathcal{K}(\!|M|\!)\, z)\, \texttt{handle}\, \lambda e.\, \mathcal{K}(\!|H|\!)\, e$$

Also, the abstraction on y may seem superfluous but without it the body of the `handle` is a value, which results in the newly installed handler being uninstalled immediately.

4.5.3 No junk

Lemma 12 (Inverseness) *1. If* $\Gamma\ ;\ \langle r, h\rangle : T \models_{\mathrm{s}} M : \neg D$ *then* $\overline{\mathcal{K}(\!|M|\!)}\llcorner\langle R, H\rangle \overset{\beta\eta}{=} R\, \delta\langle r, h\rangle.\, M$.

2. If $\Gamma\ ;\ \langle r, h\rangle : T \models_{\mathrm{s}} M : T$ *then* $\overline{\mathcal{T}(\!|M|\!)[N]} \overset{\beta\eta}{=} \delta\langle r, h\rangle.\, \overline{N}\llcorner M$.

3. If $\Gamma\ ;\ -\ \models_{\mathrm{s}} M : D$ *or* $\Gamma\ ;\ -\ \models_{\mathrm{s}} M : T \multimap \neg D$ *then* $\overline{\mathcal{D}(\!|M|\!)}\llcorner\langle R, H\rangle \overset{\beta\eta}{=} R\, M$.

4. If $\Gamma\ ;\ -\ \models_{\mathrm{s}} M : T \multimap \mathbf{R}$ *then* $\overline{\mathcal{P}(\!|M|\!)} \overset{\beta\eta}{=} M$.

5. If $\Gamma\ ;\ \langle r, h\rangle : T \models_{\mathrm{s}} M : \mathbf{R}$ *then* $\overline{\mathcal{R}(\!|M|\!)} \overset{\beta\eta}{=} \delta\langle r, h\rangle.\, M$.

Proof By induction on the derivation of the judgment:

[KABS]: Therefore $M = \lambda x.\, M'$ and $\Gamma\ ,\ x : D\ ;\ \langle r, h\rangle : T \models_{\mathrm{s}} M' : \mathbf{R}$. Hence

$$\begin{aligned}
\overline{\mathcal{K}(\!|\lambda x.\, M'|\!)}\llcorner\langle R, H\rangle &= \overline{\lambda x.\, \mathcal{R}(\!|M'|\!)}\llcorner\langle R, H\rangle \\
&= (\delta\langle r, h\rangle.\, r\, \ulcorner\lambda x.\, \mathcal{R}(\!|M'|\!)\urcorner)\llcorner\langle R, H\rangle \\
&\overset{\beta\eta}{=} R\, \ulcorner\lambda x.\, \mathcal{R}(\!|M'|\!)\urcorner \\
&= R\, \delta\langle r, h\rangle.\, \lambda x.\, \overline{\mathcal{R}(\!|M'|\!)}\llcorner\langle r, h\rangle \\
&\overset{\beta\eta}{=} R\, \delta\langle r, h\rangle.\, \lambda x.\, (\delta\langle r, h\rangle.\, M')\llcorner\langle r, h\rangle \quad \text{by the induction hypothesis} \\
&\overset{\beta\eta}{=} R\, \delta\langle r, h\rangle.\, \lambda x.\, M'
\end{aligned}$$

[KAPP]: Therefore $M = M'\llcorner N'$, $\Gamma\ ;\ -\ \models_{\mathrm{s}} M' : T \multimap \neg D$, and $\Gamma\ ;\ \langle r, h\rangle : T \models_{\mathrm{s}} N' : T$. Hence

$$
\begin{aligned}
&\overline{\mathcal{K}(\!|M' \llcorner N'|\!)} \llcorner \langle R, H\rangle \\
&= \overline{\lambda x.\, \mathcal{T}(\!|N'|\!)[\mathcal{D}(\!|M'|\!)\, x]} \llcorner \langle R, H\rangle \\
&= (\delta\langle r, h\rangle.\, r \ulcorner \lambda x.\, \mathcal{T}(\!|N'|\!)[\mathcal{D}(\!|M'|\!)\, x] \urcorner) \llcorner \langle R, H\rangle \\
&\overset{\beta\eta}{=} R \ulcorner \lambda x.\, \mathcal{T}(\!|N'|\!)[\mathcal{D}(\!|M'|\!)\, x] \urcorner \\
&= R\, \delta\langle r, h\rangle.\, \lambda x.\, \overline{\mathcal{T}(\!|N'|\!)[\mathcal{D}(\!|M'|\!)\, x]} \llcorner \langle r, h\rangle \\
&\overset{\beta\eta}{=} R\, \delta\langle r, h\rangle.\, \lambda x.\, (\delta\langle r, h\rangle.\, \overline{\mathcal{D}(\!|M'|\!)\, x} \llcorner N') \llcorner \langle r, h\rangle \qquad \text{by the induction hypothesis} \\
&\overset{\beta\eta}{=} R\, \delta\langle r, h\rangle.\, \lambda x.\, \overline{\mathcal{D}(\!|M'|\!)\, x} \llcorner N' \\
&= R\, \delta\langle r, h\rangle.\, \lambda x.\, (\delta\langle r, h\rangle.\, \overline{\mathcal{D}(\!|M'|\!)} \llcorner \langle \lambda m.\, \bar{x} \llcorner \langle \lambda n.\, m \llcorner \langle r, h\rangle\, n, h\rangle, h\rangle) \llcorner N' \\
&\overset{\beta\eta}{=} R\, \delta\langle r, h\rangle.\, \lambda x.\, (\delta\langle r, h\rangle.\, \overline{\mathcal{D}(\!|M'|\!)} \llcorner \langle \lambda m.\, \bar{x} \llcorner \langle m \llcorner \langle r, h\rangle, h\rangle, h\rangle) \llcorner N' \\
&= R\, \delta\langle r, h\rangle.\, \lambda x.\, (\delta\langle r, h\rangle.\, \overline{\mathcal{D}(\!|M'|\!)} \llcorner \langle \lambda m.\, (\delta\langle r, h\rangle.\, r \ulcorner x \urcorner) \llcorner \langle m \llcorner \langle r, h\rangle, h\rangle, h\rangle) \llcorner N' \\
&\overset{\beta\eta}{=} R\, \delta\langle r, h\rangle.\, \lambda x.\, (\delta\langle r, h\rangle.\, \overline{\mathcal{D}(\!|M'|\!)} \llcorner \langle \lambda m.\, m \llcorner \langle r, h\rangle \ulcorner x \urcorner, h\rangle) \llcorner N' \\
&= R\, \delta\langle r, h\rangle.\, \lambda x.\, (\delta\langle r, h\rangle.\, \overline{\mathcal{D}(\!|M'|\!)} \llcorner \langle \lambda m.\, m \llcorner \langle r, h\rangle\, x, h\rangle) \llcorner N' \\
&\overset{\beta\eta}{=} R\, \delta\langle r, h\rangle.\, \lambda x.\, (\delta\langle r, h\rangle.\, (\lambda m.\, m \llcorner \langle r, h\rangle\, x)\, M') \llcorner N' \qquad \text{by the induction hypothesis} \\
&\overset{\beta\eta}{=} R\, \delta\langle r, h\rangle.\, \lambda x.\, (\delta\langle r, h\rangle.\, M' \llcorner \langle r, h\rangle\, x) \llcorner N' \\
&= R\, \delta\langle r, h\rangle.\, \lambda x.\, (\delta t.\, M' \llcorner t\, x) \llcorner N' \\
&\overset{\beta\eta}{=} R\, \delta\langle r, h\rangle.\, \lambda x.\, M' \llcorner N'\, x \\
&\overset{\beta\eta}{=} R\, \delta\langle r, h\rangle.\, M' \llcorner N'
\end{aligned}
$$

[TID]: Therefore $\Gamma\,;\, \langle r, h\rangle : T \models_{\mathrm{s}} M : T = \Gamma\,;\, t : T \models_{\mathrm{s}} t : T$. Hence

$$
\begin{aligned}
\overline{\mathcal{T}(\!|t|\!)[N]} &= \overline{[][N]} \\
&= \overline{N} \\
&\overset{\beta\eta}{=} \delta t.\, \overline{N} \llcorner t
\end{aligned}
$$

[TPAIR]: Therefore $M = \langle M', N'\rangle$, $\Gamma\,;\, \langle r, h\rangle : T \models_{\mathrm{s}} M' : \neg D$, and $\Gamma\,;\, \langle r, h\rangle : T \models_{\mathrm{s}} N' : \neg D$. Hence

$$
\begin{aligned}
&\overline{\mathcal{T}(\!|\langle M', N'\rangle|\!)[N]} \\
&= \overline{(((\lambda y.\, \lambda x.\, x\, y)\, [])\, \mathtt{handle}\, (\lambda e.\, \lambda x.\, \mathcal{K}(\!|N'|\!)\, e)\, \mathcal{K}(\!|M'|\!))[N]} \\
&= \overline{((\lambda y.\, \lambda x.\, x\, y)\, N)\, \mathtt{handle}\, (\lambda e.\, \lambda x.\, \mathcal{K}(\!|N'|\!)\, e)\, \mathcal{K}(\!|M'|\!)} \\
&= \delta\langle r, h\rangle.\, \overline{((\lambda y.\, \lambda x.\, x\, y)\, N)\, \mathtt{handle}\, \lambda e.\, \lambda x.\, \mathcal{K}(\!|N'|\!)\, e} \\
&\qquad \llcorner \langle \lambda m.\, \overline{\mathcal{K}(\!|M'|\!)} \llcorner \langle \lambda n.\, m \llcorner \langle r, h\rangle\, n, h\rangle, h\rangle \\
&\overset{\beta\eta}{=} \delta\langle r, h\rangle.\, \overline{((\lambda y.\, \lambda x.\, x\, y)\, N)\, \mathtt{handle}\, \lambda e.\, \lambda x.\, \mathcal{K}(\!|N'|\!)\, e} \\
&\qquad \llcorner \langle \lambda m.\, \overline{\mathcal{K}(\!|M'|\!)} \llcorner \langle m \llcorner \langle r, h\rangle, h\rangle, h\rangle \\
&\overset{\beta\eta}{=} \delta\langle r, h\rangle.\, \overline{((\lambda y.\, \lambda x.\, x\, y)\, N)\, \mathtt{handle}\, \lambda e.\, \lambda x.\, \mathcal{K}(\!|N'|\!)\, e} \llcorner \langle \lambda m.\, m \llcorner \langle r, h\rangle\, \delta\langle r, h\rangle.\, M', h\rangle \\
&\qquad \text{by the induction hypothesis} \\
&= \delta\langle r, h\rangle.\, (\delta\langle r, h\rangle.\, \overline{((\lambda y.\, \lambda x.\, x\, y)\, N)\, \mathtt{handle}} \\
&\qquad\qquad \llcorner \langle \lambda m.\, \overline{\lambda e.\, \lambda x.\, \mathcal{K}(\!|N'|\!)\, e} \llcorner \langle \lambda n.\, m \llcorner \langle r, h\rangle\, n, h\rangle, h\rangle) \\
&\qquad \llcorner \langle \lambda m.\, m \llcorner \langle r, h\rangle\, \delta\langle r, h\rangle.\, M', h\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\overline{((\lambda y.\,\lambda x.\,x\,y)\,N)\,\texttt{handle}}\\
&\qquad\qquad \sqcup\langle \lambda m.\,\overline{\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e}\sqcup\langle m\sqcup\langle r,h\rangle,h\rangle\rangle)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{((\lambda y.\,\lambda x.\,x\,y)\,N)\,\texttt{handle}}\\
&\qquad \sqcup\langle \lambda m.\,\overline{\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e}\sqcup\langle m\sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle,h\rangle,h\rangle\\
&= \delta\langle r,h\rangle.\,\overline{((\lambda y.\,\lambda x.\,x\,y)\,N)\,\texttt{handle}}\\
&\qquad \sqcup\langle \lambda m.\,(\delta\langle r,h\rangle.\,r\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner)\\
&\qquad\qquad \sqcup\langle m\sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle,h\rangle,h\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{((\lambda y.\,\lambda x.\,x\,y)\,N)\,\texttt{handle}}\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner,h\rangle\\
&= \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,r\,\ulcorner((\lambda y.\,\lambda x.\,x\,y)\,N)\,\texttt{handle}\urcorner)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner,h\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,(\lambda m.\,m\sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner)\\
&\qquad \ulcorner((\lambda y.\,\lambda x.\,x\,y)\,N)\,\texttt{handle}\urcorner\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\ulcorner((\lambda y.\,\lambda x.\,x\,y)\,N)\,\texttt{handle}\urcorner\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&= \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,\overline{(\lambda y.\,\lambda x.\,x\,y)\,N}\sqcup\langle r,\lambda m.\,x\sqcup\langle r,h\rangle\,m\rangle)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,\overline{(\lambda y.\,\lambda x.\,x\,y)\,N}\sqcup\langle r,x\sqcup\langle r,h\rangle\rangle)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&= \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,(\delta\langle r,h\rangle.\,\overline{\lambda y.\,\lambda x.\,x\,y}\sqcup\langle \lambda m.\,\overline{N}\sqcup\langle \lambda n.\,m\sqcup\langle r,h\rangle\,n,h\rangle,h\rangle)\\
&\qquad\qquad \sqcup\langle r,x\sqcup\langle r,h\rangle\rangle)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,(\delta\langle r,h\rangle.\,\overline{\lambda y.\,\lambda x.\,x\,y}\sqcup\langle \lambda m.\,\overline{N}\sqcup\langle m\sqcup\langle r,h\rangle,h\rangle,h\rangle)\sqcup\langle r,x\sqcup\langle r,h\rangle\rangle)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&= \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,(\delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,r\,\ulcorner\lambda y.\,\lambda x.\,x\,y\urcorner)\sqcup\langle \lambda m.\,\overline{N}\sqcup\langle m\sqcup\langle r,h\rangle,h\rangle,h\rangle)\\
&\qquad\qquad \sqcup\langle r,x\sqcup\langle r,h\rangle\rangle)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,(\delta\langle r,h\rangle.\,(\lambda m.\,\overline{N}\sqcup\langle m\sqcup\langle r,h\rangle,h\rangle)\,\ulcorner\lambda y.\,\lambda x.\,x\,y\urcorner)\\
&\qquad\qquad \sqcup\langle r,x\sqcup\langle r,h\rangle\rangle)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,(\delta\langle r,h\rangle.\,\overline{N}\sqcup\langle\ulcorner\lambda y.\,\lambda x.\,x\,y\urcorner\sqcup\langle r,h\rangle,h\rangle)\sqcup\langle r,x\sqcup\langle r,h\rangle\rangle)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&= \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,(\delta\langle r,h\rangle.\,\overline{N}\sqcup\langle(\delta\langle r,h\rangle.\,\lambda y.\,\overline{\lambda x.\,x\,y}\sqcup\langle r,h\rangle)\sqcup\langle r,h\rangle,h\rangle)\\
&\qquad\qquad \sqcup\langle r,x\sqcup\langle r,h\rangle\rangle)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,(\delta\langle r,h\rangle.\,\overline{N}\sqcup\langle\lambda y.\,\overline{\lambda x.\,x\,y}\sqcup\langle r,h\rangle,h\rangle)\sqcup\langle r,x\sqcup\langle r,h\rangle\rangle)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&= \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,(\delta\langle r,h\rangle.\,\overline{N}\sqcup\langle\lambda y.\,(\delta\langle r,h\rangle.\,r\,\ulcorner\lambda x.\,x\,y\urcorner)\sqcup\langle r,h\rangle,h\rangle)\\
&\qquad\qquad \sqcup\langle r,x\sqcup\langle r,h\rangle\rangle)\\
&\qquad \sqcup\langle \lambda m.\,m\sqcup\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner
\end{aligned}
$$

$$
\begin{aligned}
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,(\delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,r\,\ulcorner\lambda x.\,x\,y\urcorner,h\rangle)\llcorner\langle r,x\llcorner\langle r,h\rangle\rangle)\\
&\qquad\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,(\delta\langle r,h\rangle.\,\lambda x.\,\overline{N}\llcorner\langle\lambda y.\,r\,\ulcorner\lambda x.\,x\,y\urcorner,x\llcorner\langle r,h\rangle\rangle)\\
&\qquad\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,(\lambda x.\,\overline{N}\llcorner\langle\lambda y.\,(\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M')\,\ulcorner\lambda x.\,x\,y\urcorner\\
&\qquad\qquad,x\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle)\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,(\lambda x.\,\overline{N}\llcorner\langle\lambda y.\,\ulcorner\lambda x.\,x\,y\urcorner\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M'\\
&\qquad\qquad,x\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle)\,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,\ulcorner\lambda x.\,x\,y\urcorner\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M'\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,(\delta\langle r,h\rangle.\,\lambda x.\,\overline{x\,y}\llcorner\langle r,h\rangle)\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M'\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,(\lambda x.\,\overline{x\,y}\llcorner\langle r,h\rangle)\,\delta\langle r,h\rangle.\,M'\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,(\lambda x.\,(\delta\langle r,h\rangle.\,\bar{x}\llcorner\langle\lambda m.\,\bar{y}\llcorner\langle\lambda n.\,m\llcorner\langle r,h\rangle\,n,h\rangle,h\rangle)\llcorner\langle r,h\rangle)\\
&\qquad\qquad\delta\langle r,h\rangle.\,M'\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,(\lambda x.\,(\delta\langle r,h\rangle.\,\bar{x}\llcorner\langle\lambda m.\,\bar{y}\llcorner\langle m\llcorner\langle r,h\rangle,h\rangle,h\rangle)\llcorner\langle r,h\rangle)\,\delta\langle r,h\rangle.\,M'\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,(\lambda x.\,\bar{x}\llcorner\langle\lambda m.\,\bar{y}\llcorner\langle m\llcorner\langle r,h\rangle,h\rangle,h\rangle)\,\delta\langle r,h\rangle.\,M'\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,(\lambda x.\,(\delta\langle r,h\rangle.\,r\,\ulcorner x\urcorner)\llcorner\langle\lambda m.\,\bar{y}\llcorner\langle m\llcorner\langle r,h\rangle,h\rangle,h\rangle)\,\delta\langle r,h\rangle.\,M'\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,(\lambda x.\,(\lambda m.\,\bar{y}\llcorner\langle m\llcorner\langle r,h\rangle,h\rangle)\,\ulcorner x\urcorner)\,\delta\langle r,h\rangle.\,M'\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,(\lambda x.\,\bar{y}\llcorner\langle\ulcorner x\urcorner\llcorner\langle r,h\rangle,h\rangle)\,\delta\langle r,h\rangle.\,M'\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,(\lambda x.\,\bar{y}\llcorner\langle x\llcorner\langle r,h\rangle,h\rangle)\,\delta\langle r,h\rangle.\,M'\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,\bar{y}\llcorner\langle(\delta\langle r,h\rangle.\,M')\llcorner\langle r,h\rangle,h\rangle\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,\bar{y}\llcorner\langle M',h\rangle,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,(\delta\langle r,h\rangle.\,r\,\ulcorner y\urcorner)\llcorner\langle M',h\rangle\\
&\qquad\qquad,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,M'\,\ulcorner y\urcorner,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle\lambda y.\,M'\,y,\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M',\ulcorner\lambda e.\,\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M',(\delta\langle r,h\rangle.\,\lambda e.\,\overline{\lambda x.\,\mathcal{K}(\!|N'|\!)\,e}\llcorner\langle r,h\rangle)\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M',\lambda e.\,\overline{\lambda x.\,\mathcal{K}(\!|N'|\!)\,e}\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M',\lambda e.\,(\delta\langle r,h\rangle.\,r\,\ulcorner\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner)\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M',h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M',\lambda e.\,(\lambda m.\,m\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M')\,\ulcorner\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,\ulcorner\lambda x.\,\mathcal{K}(\!|N'|\!)\,e\urcorner\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M'\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,(\delta\langle r,h\rangle.\,\lambda x.\,\overline{\mathcal{K}(\!|N'|\!)\,e}\llcorner\langle r,h\rangle)\llcorner\langle r,h\rangle\,\delta\langle r,h\rangle.\,M'\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,(\lambda x.\,\overline{\mathcal{K}(\!|N'|\!)\,e}\llcorner\langle r,h\rangle)\,\delta\langle r,h\rangle.\,M'\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,\overline{\mathcal{K}(\!|N'|\!)\,e}\llcorner\langle r,h\rangle\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,(\delta\langle r,h\rangle.\,\overline{\mathcal{K}(\!|N'|\!)}\llcorner\langle\lambda m.\,\overline{e}\llcorner\langle\lambda n.\,m\llcorner\langle r,h\rangle\,n,h\rangle,h\rangle)\llcorner\langle r,h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,(\delta\langle r,h\rangle.\,\overline{\mathcal{K}(\!|N'|\!)}\llcorner\langle\lambda m.\,\overline{e}\llcorner\langle m\llcorner\langle r,h\rangle,h\rangle,h\rangle)\llcorner\langle r,h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,\overline{\mathcal{K}(\!|N'|\!)}\llcorner\langle\lambda m.\,\overline{e}\llcorner\langle m\llcorner\langle r,h\rangle,h\rangle,h\rangle\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,\overline{\mathcal{K}(\!|N'|\!)}\llcorner\langle\lambda m.\,(\delta\langle r,h\rangle.\,r\,\ulcorner e\urcorner)\llcorner\langle m\llcorner\langle r,h\rangle,h\rangle,h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,\overline{\mathcal{K}(\!|N'|\!)}\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,\ulcorner e\urcorner,h\rangle\rangle\\
&= \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,\overline{\mathcal{K}(\!|N'|\!)}\llcorner\langle\lambda m.\,m\llcorner\langle r,h\rangle\,e,h\rangle\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,(\lambda m.\,m\llcorner\langle r,h\rangle\,e)\,\delta\langle r,h\rangle.\,N'\rangle \quad \text{by the induction hypothesis}\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,(\delta\langle r,h\rangle.\,N')\llcorner\langle r,h\rangle\,e\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', \lambda e.\,N'\,e\rangle\\
&\overset{\beta\eta}{=} \delta\langle r,h\rangle.\,\overline{N}\llcorner\langle M', N'\rangle
\end{aligned}
$$

[DID]: Therefore $M = x$. Hence

$$
\begin{aligned}
\overline{\mathcal{D}(\!|x|\!)}\llcorner\langle R,H\rangle &= \overline{x}\llcorner\langle R,H\rangle\\
&= (\delta\langle r,h\rangle.\,r\,\ulcorner x\urcorner)\llcorner\langle R,H\rangle\\
&\overset{\beta\eta}{=} R\,\ulcorner x\urcorner\\
&= R\,x
\end{aligned}
$$

[DABS]: Therefore $M = \delta\langle r,h\rangle.\,M'$ and $\Gamma\,;\,\langle r,h\rangle : T \vDash_{\mathrm{s}} M' : \neg D$. Hence

$$
\begin{aligned}
\overline{\mathcal{D}(\!|\delta\langle r,h\rangle.\,M'|\!)}\llcorner\langle R,H\rangle &= \overline{\mathcal{K}(\!|M'|\!)}\llcorner\langle R,H\rangle\\
&\overset{\beta\eta}{=} R\,\delta\langle r,h\rangle.\,M' \qquad \text{by the induction hypothesis}
\end{aligned}
$$

[DPROJ0]: Therefore $M = \pi_0$. Hence

$$
\begin{aligned}
\overline{\mathcal{D}(\!|\pi_0|\!)}\llcorner\langle R,H\rangle &= \overline{\lambda x.\,x}\llcorner\langle R,H\rangle\\
&= (\delta\langle r,h\rangle.\,r\,\ulcorner\lambda x.\,x\urcorner)\llcorner\langle R,H\rangle\\
&\overset{\beta\eta}{=} R\,\ulcorner\lambda x.\,x\urcorner\\
&= R\,\delta\langle r,h\rangle.\,\lambda x.\,\overline{x}\llcorner\langle r,h\rangle\\
&= R\,\delta\langle r,h\rangle.\,\lambda x.\,(\delta\langle r,h\rangle.\,r\,\ulcorner x\urcorner)\llcorner\langle r,h\rangle\\
&\overset{\beta\eta}{=} R\,\delta\langle r,h\rangle.\,\lambda x.\,r\,\ulcorner x\urcorner\\
&= R\,\delta\langle r,h\rangle.\,\lambda x.\,r\,x\\
&\overset{\beta\eta}{=} R\,\delta\langle r,h\rangle.\,r\\
&= R\,\delta t.\,\pi_0\llcorner t\\
&\overset{\beta\eta}{=} R\,\pi_0
\end{aligned}
$$

[DPROJ1]: Therefore $M = \pi_1$. Hence

$$\begin{aligned}
\overline{\mathcal{D}(\!|\pi_1|\!)}\llcorner\langle R, H\rangle &= \overline{\mathtt{raise}}\llcorner\langle R, H\rangle \\
&= (\delta\langle r, h\rangle.\, r\,\ulcorner\mathtt{raise}\urcorner)\llcorner\langle R, H\rangle \\
&\overset{\beta\eta}{=} R\,\ulcorner\mathtt{raise}\urcorner \\
&= R\,\delta\langle r, h\rangle.\, \lambda x.\, h\,x \\
&\overset{\beta\eta}{=} R\,\delta\langle r, h\rangle.\, h \\
&= R\,\delta t.\, \pi_1\llcorner t \\
&\overset{\beta\eta}{=} R\,\pi_1
\end{aligned}$$

[DFOLD]: Therefore $\Gamma\ ;\ -\ \vDash_{\mathrm{s}} M : T \multimap \neg D$. Hence

$$\overline{\mathcal{D}(\!|M|\!)}\llcorner\langle R, H\rangle \overset{\beta\eta}{=} R\,M \quad \text{by the induction hypothesis}$$

[DUNFOLD]: Therefore $\Gamma\ ;\ -\ \vDash_{\mathrm{s}} M : D$. Hence

$$\overline{\mathcal{D}(\!|M|\!)}\llcorner\langle R, H\rangle \overset{\beta\eta}{=} R\,M \quad \text{by the induction hypothesis}$$

[PABS]: Therefore $M = \delta\langle r, h\rangle.\, M'$ and $\Gamma\ ;\ \langle r, h\rangle : T \vDash_{\mathrm{s}} M' : \mathbf{R}$. Hence

$$\begin{aligned}
\overline{\mathcal{P}(\!|\delta\langle r, h\rangle.\, M'|\!)} &= \overline{\mathcal{R}(\!|M'|\!)} \\
&\overset{\beta\eta}{=} \delta\langle r, h\rangle.\, M' \quad \text{by the induction hypothesis}
\end{aligned}$$

[RAPPK]: Therefore $M = M'\,N'$, $\Gamma\ ;\ \langle r, h\rangle : T \vDash_{\mathrm{s}} M' : \neg D$, and $\Gamma\ ;\ -\ \vDash_{\mathrm{s}} N' : D$. Hence

$$\begin{aligned}
&\overline{\mathcal{R}(\!|M'\,N'|\!)} \\
&= \overline{\mathcal{K}(\!|M'|\!)\,\mathcal{D}(\!|N'|\!)} \\
&= \delta\langle r, h\rangle.\, \overline{\mathcal{K}(\!|M'|\!)}\llcorner\langle \lambda m.\, \overline{\mathcal{D}(\!|N'|\!)}\llcorner\langle \lambda n.\, m\llcorner\langle r, h\rangle\, n, h\rangle, h\rangle \\
&\overset{\beta\eta}{=} \delta\langle r, h\rangle.\, \overline{\mathcal{K}(\!|M'|\!)}\llcorner\langle \lambda m.\, \overline{\mathcal{D}(\!|N'|\!)}\llcorner\langle m\llcorner\langle r, h\rangle, h\rangle, h\rangle \\
&\overset{\beta\eta}{=} \delta\langle r, h\rangle.\, \overline{\mathcal{K}(\!|M'|\!)}\llcorner\langle \lambda m.\, m\llcorner\langle r, h\rangle\, N', h\rangle && \text{by the induction hypothesis} \\
&\overset{\beta\eta}{=} \delta\langle r, h\rangle.\, (\lambda m.\, m\llcorner\langle r, h\rangle\, N')\,\delta\langle r, h\rangle.\, M' && \text{by the induction hypothesis} \\
&\overset{\beta\eta}{=} \delta\langle r, h\rangle.\, (\delta\langle r, h\rangle.\, M')\llcorner\langle r, h\rangle\, N' \\
&\overset{\beta\eta}{=} \delta\langle r, h\rangle.\, M'\,N'
\end{aligned}$$

[RAPPP]: Therefore $M = M'\llcorner N'$, $\Gamma\ ;\ -\ \vDash_{\mathrm{s}} M' : T \multimap \mathbf{R}$, and $\Gamma\ ;\ \langle r, h\rangle : T \vDash_{\mathrm{s}} N' : T$. Hence

$$\begin{aligned}
\overline{\mathcal{R}(\!|M'\llcorner N'|\!)} &= \overline{\mathcal{T}(\!|N'|\!)[\mathcal{P}(\!|M'|\!)]} \\
&\overset{\beta\eta}{=} \delta\langle r, h\rangle.\, \overline{\mathcal{P}(\!|M'|\!)}\llcorner N' \quad \text{by the induction hypothesis} \\
&\overset{\beta\eta}{=} \delta\langle r, h\rangle.\, M'\llcorner N' \quad \text{by the induction hypothesis}
\end{aligned}$$

□

Proof [Proposition 9] Let $N = \mathcal{P}(\!|M|\!)$ and the result follows by Lemma 12. □

4.6 Conclusion

In this chapter we have seen that the treatment and results of procedures extends to a source language with exceptions. This demonstration that restricted typing of CPS applies to a language with quite expressive control behavior begins to demonstrate the veracity of the technique.

Technically, the main point is that the interpretation of procedures:

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to \mathbf{R})}_{\substack{\text{return}\\ \text{continuation}}}\ \&\ \underbrace{(D \to \mathbf{R})}_{\substack{\text{handler}\\ \text{continuation}}}\ \multimap\ \underbrace{D \to \mathbf{R}}_{\substack{\text{call}\\ \text{continuation}}}$$

uses linear logic's additive product to allow one of two continuations to be "used," but not both. This is our first encounter with the more general issue that it is not *continuations*, but *control contexts* such as $\&$-pairs of continuations, which should have their use restricted.

Chapter 5

Delimited Continuations, and Control Contexts versus Continuations

---

The main goal of this chapter is to explain the uncommon but crucial distinction we make between control contexts and continuations. This distinction is necessary since the role each parameter of a semantics (an environment, a store, a current continuation, etc.) plays, rather than its type, is of primary concern. Hence we focus on restricted use of control contexts, which are defined by the role they play, rather than continuations, which are defined independently as destinations of jumps. To demonstrate the importance of this distinction, we give an interpretation of first-class continuations (`call/cc` and `abort`) which uses continuations linearly, but control contexts nonlinearly. What this demonstrates is that care must be taken to ensure that the entire control context is used linearly by an interpretation, or else linearity says virtually nothing. In other words, linear typing is a tool but not itself a solution.

The key aspect of this interpretation is that control contexts are represented not as continuations, as is usual, but as a continuation (which is used linearly) together with a delimited continuation (which is used nonlinearly). We will discuss them more fully later, but in short a delimited continuation represents *part* of a computation, rather than a computation "to the end" as an undelimited continuation represents. As several later interpretations are based on delimited continuations, we first introduce them in their own right.

Unlike most presentations of delimited continuations (similar to, or also known as: composable continuations, partial continuations, functional continuations, and subcontinuations, see Section 5.5), we consider them only in the target language, independent of any delimited continuation control construct in the source. Since we do not consider any such operations, we never need to *compose* delimited continuations (as is done in e.g. [DF90]), and so given that the ability to do so is one of the primary characteristics of delimited continuations, our use of them here is somewhat degenerate. By giving a delimited continuation interpretation of λ-calculus, we introduce the technical formulation of delimited continuations we use, and present some related techniques and ideas we will use in Section 5.4 and later chapters.

5.1 Delimited Continuations

When a continuation is invoked, it performs some computation and then invokes its parent continuation, which in turn performs some computation and then invokes its parent continuation, and so on. So, conceptually, there are two parts to a continuation: the "first part," and the "rest part" (the parent continuation). (Recall that we have come across this intuition already: it was crucial to the DS transformations of Section 3.6.3 and Section 4.5.2.) More generally, a continuation can be split not only at the parent continuation, but also at the parent's parent, some number

of times removed. In this case, the continuation is split in to a delimited continuation which is the composition of *n*-many "first parts", and an undelimited continuation which is the parent's parent, *n*-times removed.

For a concrete example of some delimited continuations (what they *are*, but not *how* to obtain them), consider the following Scheme program which uses delimited continuations explicitly:

```
(define (fact-dcps n k t)
  (if (= n 0)
      (k 1 t)
      (fact-dcps
       (- n 1)
       (lambda (x t) (k (* n x) t))
       t)))

(define (fact-cps' n k)
  (fact-dcps n (lambda (x t) (t x)) k))
```

To avoid conjuring this program out of thin air, we show the analogous program in standard CPS, which uses undelimited continuations:

```
(define (fact-cps n k)
  (if (= n 0)
      (k 1)
      (fact-cps
       (- n 1)
       (lambda (x) (k (* n x))))))
```

In the former program, each continuation takes an extra argument: the toplevel continuation. That is, each undelimited continuation in the latter program has been split into a delimited continuation together with the toplevel (undelimited) continuation. Thus far technically we have ignored such potential divisions and simply treated continuations atomically, but we now consider these "first parts," that is, delimited continuations, separately.

Note that while we relate these programs using η-equivalence, we do not mean to imply that η is generally the way to transit from programs manipulating undelimited continuations to programs using delimited continuations. Instead, the CPS transformation should be applied the CPS version to yield the delimited continuation-passing version [DF90]. So, while the previous example suffices to demonstrate some inhabitants of the type of delimited continuations, the following (taken from [BBD04b], but simplified to the original problem in [Dan89], and written in Scheme) is a more appropriate example of delimited continuations used "in anger," where they actually buy something operationally rather than just reorganize code:

```
(define (prefixes-cps xs t)
  (letrec ((visit
            (lambda (xs d k)
              (if (not (pair? xs))
                  (k '())
                  (let ((c (lambda (vs h) (d (cons (car xs) vs) h))))
                    (c '() (lambda (vs)
                             (visit (cdr xs) c
                                    (lambda (vss) (k (cons vs vss))))))))))))
    (visit xs (lambda (vs k) (k vs)) t)))
```

This function returns a list of all the prefixes of the argument list, so `(prefixes-cps '(1 2 3 4) t)` passes `((1) (1 2) (1 2 3) (1 2 3 4))` to `t`. Note that the auxiliary `visit` function's second argument is a delimited continuation, and its third argument is an undelimited continuation.

In this program, use of delimited continuations is much more significant as it allows sharing of the control which constructs the common parts of the list prefixes. Each delimited continuation represents a list prefix and, when invoked with a list and a continuation, constructs the prefix onto the given list and then invokes the continuation on the prefixed list. The key here is how each delimited continuation representing a certain prefix is shared by all the delimited continuations representing longer prefixes. This is accomplished in the code by invoking delimited continuation `c` with a continuation which itself refers to `c`. In this way, `c` is used once to construct the prefix it represents, and then later by its continuation, potentially many times, to construct the longer prefixes. See [BBD04b] for a full discussion and explanation.

Intuitively, a *delimited continuation*, like a continuation, is the destination of a jump with arguments,[1] but unlike a continuation, a delimited continuation represents only part of a computation while the remainder is represented by a continuation argument which the delimited continuation accepts.[2] So invoking either a continuation or a delimited continuation results in performing some computation and then invoking a continuation. The difference is that with a continuation, the second continuation is an indivisible part of the first, while with a delimited continuation, the continuation is an argument of the delimited continuation.

For example, the undelimited continuation involved in the computation of `(fact-cps 3 k)`

```
(lambda (x)
  ((lambda (x)
     ((lambda (x)
        (k
         (* 3 x)))
      (* 2 x)))
   (* 1 x)))
```

$\stackrel{\beta\eta}{=}$ `(lambda (x) (k (* 3 (* 2 (* 1 x)))))`

contains its parent/toplevel continuation, `k`, while the corresponding delimited continuation involved in the computation of `(fact-cps' 3 k)`

```
(lambda (x t)
  ((lambda (x t)
     ((lambda (x t)
        ((lambda (x t) (t x))
         (* 3 x) t))
      (* 2 x) t))
   (* 1 x) t))
```

$\stackrel{\beta\eta}{=}$ `(lambda (x t) (t (* 3 (* 2 (* 1 x)))))`

contains no continuation, and instead accepts the toplevel continuation as an argument, `t`. Also note how the "first parts" of three continuations are composed together above, particularly evidently in the β-contracted versions.

At the level of types, for a continuation of type $S \to \mathbf{R}$ which contains a continuation of type $T \to \mathbf{R}$

$$\Gamma\,, k : T \to \mathbf{R} \vdash M : S \to \mathbf{R}$$

[1] Note, however, that in presentations of delimited continuations in the source language, when invoked, a delimited continuation is *composed* with the continuation of the invocation site, and so the continuation argument of a delimited continuation is implicitly taken to be the current continuation of the invocation site. In this case, invoking a delimited continuation has no abortive effect since the current continuation remains intact, and hence is, in a way, unlike a jump.

[2] In [Dan88, DF89, DF90, DF92], what we call "delimited continuations" are termed "continuations," and our "continuations" are termed "meta-continuations."

the corresponding delimited continuation will have type $S \to (T \to \mathbf{R}) \to \mathbf{R}$ and will not contain any (free) continuations:

$$\Gamma \vdash \lambda x.\, \lambda k.\, M\, x : S \to (T \to \mathbf{R}) \to \mathbf{R}$$

So the implicit dependence of a continuation on another continuation through the context is changed to the explicit dependence of a delimited continuation on an argument continuation. The crucial point of our use of delimited continuations is that decomposing a continuation into delimited and undelimited portions allows the two pieces to be treated separately in the type system. So if the two pieces are manipulated in different ways, then more accurate restricted typings may be possible.

Note that while delimited continuations are continuations, conceptually (they are destinations of jumps) and at the level of types (they are maps to the result type), in discussions where both delimited and undelimited continuations are involved, we use the term "continuation" to refer not to either an undelimited or delimited continuation, but to only the undelimited continuations under consideration. This is entirely a matter of convenience for discussion, since with an abstract result type, one never knows if a continuation is actually delimited in some way hidden by the abstractness of the result type.

5.2 Refined Delimited Continuation Interpretation of Procedures

We introduce our use of delimited continuations by giving an alternate presentation of Chapter 3's interpretation of the untyped λ-calculus, one which primarily manipulates delimited continuations rather than (undelimited) continuations. Recall that in Section 3.3, value terms (procedures) were interpreted with the type (where we have renamed $\mathbf{R}$ to $\mathbf{R}'$)

$$\mu D.\ \underbrace{(D \to \mathbf{R}')}_{\substack{\text{return}\\ \text{continuation}}} \multimap \underbrace{D \to \mathbf{R}'}_{\substack{\text{call}\\ \text{continuation}}}$$

and terms with the type

$$\underbrace{(D \to \mathbf{R}')}_{\substack{\text{return}\\ \text{continuation}}} \multimap \mathbf{R}'$$

Since $\mathbf{R}'$ is abstract, the transformation remains correct with any result type. In particular, partially specifying the result type to

$$\mathbf{R}' \stackrel{\text{def}}{=} \underbrace{(D \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{R}$$

leads to an interpretation of procedures using the type[3]

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to \underbrace{(D \to \mathbf{R})}_{\text{toplevel continuation}} \multimap \mathbf{R})}_{\substack{\text{return}\\ \text{delimited continuation}}} \multimap \underbrace{D \to \underbrace{(D \to \mathbf{R})}_{\text{toplevel continuation}} \multimap \mathbf{R}}_{\substack{\text{call}\\ \text{delimited continuation}}}$$

[3]We use a combination of continuation-first and continuation-second styles here to remain consistent with our other interpretations while still being able to benefit from η-reductions to simplify terms which do not manipulate the toplevel continuation.

and an interpretation of terms using the type

$$\underbrace{(D \to \underbrace{(D \to \mathbf{R})}_{\text{toplevel continuation}} \multimap \mathbf{R})}_{\substack{\text{return}\\ \text{delimited continuation}}} \multimap \underbrace{(D \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{R}$$

In this interpretation, the call and return continuations of the old interpretation are replaced with delimited continuations. The usual return continuation can be obtained from a return delimited continuation R and toplevel continuation T by

$$\lambda x.\, R\, x\, T$$

So this new interpretation of procedures is simply explicating how a return continuation is made up of the toplevel continuation together with a delimited continuation. In other words, this interpretation uses a different representation of control contexts: the control context of a procedure call is now represented not directly by a return continuation, but by a return delimited continuation and the toplevel continuation.

Another point to note about this interpretation is that CPS procedures accept the toplevel continuation and then pass it on to the return delimited continuation. So the toplevel continuation is now essentially a piece of global state, although it is never modified. The toplevel continuation is passed to each delimited continuation so that it may terminate the computation, but a delimited continuation does not "own" the toplevel continuation like an undelimited continuation owns its parent continuation: a delimited continuation and the toplevel continuation are divisible and independently manipulable, unlike an undelimited continuation which indivisibly contains the toplevel continuation. This opens up the possibility of having several delimited continuations around without duplicating the toplevel continuation, which is precisely what we need in Chapter 9.

Since all we have done is further specify the result type, the transformation on terms, shown in Figure 5.1, is simply a linear η-expansion of the transformation in Section 3.3. Hence Soundness and Adequacy also hold for this interpretation.

The interpretation of toplevel programs using the type

$$\underbrace{(D \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{R}$$

shown in Figure 5.1, simply installs the "empty" delimited continuation (the *unit* of delimited continuation composition) as the return delimited continuation since, at toplevel, the return continuation essentially is the toplevel continuation. While not as immediate as for the transformation on terms, Soundness and Adequacy results for this interpretation are straightforward extensions of those in Chapter 3.

Figure 5.1 Refined Delimited Continuation Interpretation of Untyped λ-calculus

Value Terms

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to (D \to \mathbf{R}) \multimap \mathbf{R})}_{\text{return}} \multimap \underbrace{D \to (D \to \mathbf{R}) \multimap \mathbf{R}}_{\text{call}}$$

$$\ulcorner x \urcorner \stackrel{\text{def}}{=} x$$
$$\ulcorner \lambda x.\, M \urcorner \stackrel{\text{def}}{=} \delta r.\ \lambda x.\, \overline{M} \llcorner r \lrcorner \stackrel{\beta\eta}{=} \delta r.\ \lambda x.\, \delta t.\, \overline{M} \llcorner r \lrcorner t$$

Terms

$$\underbrace{(D \to (D \to \mathbf{R}) \multimap \mathbf{R})}_{\text{return}} \multimap \underbrace{(D \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{R}$$

$$\overline{V} \stackrel{\text{def}}{=} \delta r.\, r \ulcorner V \urcorner \qquad \stackrel{\beta\eta}{=} \delta r.\, \delta t.\, r \ulcorner V \urcorner \lrcorner t$$
$$\overline{M\,N} \stackrel{\text{def}}{=} \delta r.\, \overline{M} \llcorner (\lambda m.\, \overline{N} \llcorner (\lambda n.\, m \llcorner r\, n \lrcorner) \lrcorner) \stackrel{\beta\eta}{=} \delta r.\, \delta t.\, \overline{M} \llcorner (\lambda m.\, \delta t.\, \overline{N} \llcorner (\lambda n.\, \delta t.\, m \llcorner r\, n \lrcorner t) \lrcorner t) \lrcorner t \qquad m \notin \mathrm{fi}(N)$$

Programs

$$\underbrace{(D \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{R}$$

$$\lfloor M \rfloor \stackrel{\text{def}}{=} \delta t.\, \overline{M} \llcorner (\lambda m.\, \delta t.\, t\, m) \lrcorner t$$

5.3 Control Contexts versus Continuations

In the preceding chapters we have given refined continuation semantics by imposing a discipline of linearly used control contexts. In doing so we have made an uncommon distinction between control contexts and continuations, but so far we have not provided much justification for this distinction. In short, we distinguish control contexts and continuations because trying to provide refined continuation semantics by imposing a discipline of linearly used continuations is not very meaningful in general. The interpretation of exceptions, in which control contexts are &-pairs of continuations, contained the first hints that a discipline of linearly used continuations might not be quite right, but in Section 5.4 we provide much more convincing evidence in the form of an interpretation of `call/cc` and `abort` which uses continuations, but not control contexts, linearly. The conclusion we draw from this is that only restricting the use of continuations, and not worrying about control contexts, does not necessarily constrain the control behavior of the source language.

A related point is that the syntactic form of a type, and hence whether it is a type of continuations or not, does not indicate whether its elements ought to be used linearly or nonlinearly.[4] Instead it is the role played by the elements which is significant. For instance, in Section 3.3, procedures are interpreted with a type of continuation transformers

$$(D \to \mathbf{R}) \multimap D \to \mathbf{R}$$

[4]Our focus on types as opposed to abstract data types is immaterial. That is, the control context ADT and continuation ADT would need to be distinguished, and the signature of the control context ADT would depend upon the language under consideration, while the continuation ADT's signature need not.

and, in the previous section, delimited continuations have type

$$D \to (D \to \mathbf{R}) \multimap \mathbf{R}$$

We could easily have represented delimited continuations with a type of continuation transformers, it is merely a technical convenience to use the type above instead. That is, if D referred to the same type in both these types, they would be isomorphic. Note that this is a significant qualification, however, since if delimited rather than undelimited continuations are used, then the type of procedures *must* be likewise "bumped up." So procedures and delimited continuations may appear quite similar at the level of types (though never identical), but it is important to appreciate the very different roles they play. Continuation transformers were used to interpret source procedures, and hence they were part of the data context, and used nonlinearly. Delimited continuations, on the other hand, were used to represent part of the control context in Section 5.1, and were used linearly.

5.4 Delimited Continuation Interpretation of First-Class Continuations

We now present an interpretation of first-class continuations, as employed in the example in Section 1.2. The key aspect of this interpretation, as mentioned earlier, is that control contexts are represented not as continuations, as is usual, but as a continuation (which is used linearly) together with a delimited continuation (which is used nonlinearly). The types used to interpret source value terms

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to (D \to \mathbf{R}) \multimap \mathbf{R})}_{\substack{\text{return}\\ \text{delimited continuation}}} \to \underbrace{D \to (D \to \mathbf{R}) \multimap \mathbf{R}}_{\substack{\text{call}\\ \text{delimited continuation}}}$$

and terms

$$\underbrace{(D \to (D \to \mathbf{R}) \multimap \mathbf{R})}_{\substack{\text{return}\\ \text{delimited continuation}}} \to \underbrace{(D \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{R}$$

are the same as those used in Section 5.2 except that now delimited continuations may be used nonlinearly, but (undelimited) continuations must still be used linearly. The transformation of terms, shown in Figure 5.2, is standard except for the clause for `abort`, since none of the others manipulate the toplevel continuation and hence are insensitive to changing from a return continuation to a return delimited continuation. For `abort`, the standard transformation would be

$$\ulcorner \texttt{abort} \urcorner = \lambda r.\, \lambda x.\, x$$

which violates the abstractness of the result type by requiring $\mathbf{R} = D$. Less drastically, at least an injection from D into $\mathbf{R}$ is needed. Using delimited continuations allows a transformation

$$\ulcorner \texttt{abort} \urcorner \stackrel{\text{def}}{=} \lambda r.\, \lambda x.\, \delta t.\, t\, x$$

which instead uses the toplevel continuation to inject elements of D into $\mathbf{R}$, preserving abstractness.

Finally, toplevel programs are interpreted in Figure 5.2 using the type

$$\underbrace{(D \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{R}$$

Figure 5.2 Delimited Continuation Interpretation of First-Class Continuations

Value Terms

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to (D \to \mathbf{R}) \multimap \mathbf{R})}_{\text{return}} \to \underbrace{D \to (D \to \mathbf{R}) \multimap \mathbf{R}}_{\text{call}}$$

$$\begin{aligned}
\ulcorner x \urcorner &\stackrel{\text{def}}{=} x \\
\ulcorner \lambda x.\, M \urcorner &\stackrel{\text{def}}{=} \lambda r.\, \lambda x.\, \overline{M}\, r \qquad \stackrel{\beta\eta}{=} \lambda r.\, \lambda x.\, \delta t.\, \overline{M}\, r \llcorner t \qquad && r \notin \mathsf{fi}(\lambda x.\, M) \\
\ulcorner \texttt{call/cc} \urcorner &\stackrel{\text{def}}{=} \lambda r.\, \lambda x.\, x\, r\, \lambda s.\, r \stackrel{\beta\eta}{=} \lambda r.\, \lambda x.\, \delta t.\, x\, r\, (\lambda s.\, \lambda x.\, \delta t.\, r\, x \llcorner t) \llcorner t \\
\ulcorner \texttt{abort} \urcorner &\stackrel{\text{def}}{=} \lambda r.\, \lambda x.\, \delta t.\, t\, x
\end{aligned}$$

Terms

$$\underbrace{(D \to (D \to \mathbf{R}) \multimap \mathbf{R})}_{\text{return}} \to \underbrace{(D \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{R}$$

$$\begin{aligned}
\overline{V} &\stackrel{\text{def}}{=} \lambda r.\, r\, \ulcorner V \urcorner \qquad \stackrel{\beta\eta}{=} \lambda r.\, \delta t.\, r\, \ulcorner V \urcorner \llcorner t && r \notin \mathsf{fi}(V) \\
\overline{M\,N} &\stackrel{\text{def}}{=} \lambda r.\, \overline{M}\, \lambda m.\, \overline{N}\, (m\, r) \stackrel{\beta\eta}{=} \lambda r.\, \delta t.\, \overline{M}\, (\lambda m.\, \delta t.\, \overline{N}\, (\lambda n.\, \delta t.\, m\, r\, n \llcorner t) \llcorner t) \llcorner t && m, r \notin \mathsf{fi}(M\,N)
\end{aligned}$$

Programs

$$\underbrace{(D \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{R}$$

$$\llcorner M \lrcorner \stackrel{\text{def}}{=} \delta t.\, \overline{M}\, (\lambda m.\, \delta t.\, t\, m) \llcorner t$$

5.5 Background and Related Work

The first incarnation of delimited continuations is probably Stoy and Strachey's *Run[]* and *Finish[]* routines in OS6 [SS72]. Roughly, *Run[]* specifies where the current continuation is split into delimited and undelimited continuations, and *Finish[]* discards the delimited continuation.

While people have been programming in CPS with essentially delimited continuations since the early days during the 1970's and 1980's, Felleisen [Fel87] defined the first DS operator for delimiting control. This definition, and those in the resulting line of research [Fel88, FFDM87, FWFD88, HDA94, Joh87, JD88, SF90], are independent of CPS, and are explained in terms of additional notions, for instance, by an algebra of activation frames [FWFD88]. Danvy and Filinski [Dan88, DF89, DF90, DF92] developed alternate control operators where delimited continuations are a natural generalization of CPS itself.[5] Linking the two lines, Danvy *et. al.* [DN01, Dan04] have shown that the representations of delimited continuations in the former line (for instance, evaluation contexts) can be mechanically obtained as defunctionalized versions of the representation of the latter line (continuation functions). Also, Shan [Sha04] has shown that Felleisen's operators are macro-expressible by Danvy and Filinski's, by using recursive continuations, but not undelimited continuations or state.

Apart from the refinement with linearity, our presentation of delimited continuations in the target language, and the interpretation here, follows from Danvy and Filinski's. This approach

[5] A generalization which, as Danvy and Filinski note [DF90], can be seen as a realization of the idea alluded to in the footnote on page 9 of [SW74].

is the simpler, more appropriate fit for our purposes, and also results in static-binding of control delimiters, rather than the dynamic-binding based methods of determining where to split the current continuation found in the other line of development.

Like us, Friedman and Sabry [FS02] also use delimited continuations only as a tool in the target language, but toward different ends. Thielecke [Thi03] uses delimited continuations in the target language to interface between restricted and unrestricted continuation-passing in a mixed linear/non-linear CPS transformation.

5.6 Conclusion

In this chapter we first saw how to adapt the interpretation of procedures to one representing the control context with delimited, rather than standard undelimited, continuations:

$$D \stackrel{\text{def}}{=} \mu D.\ (D \to \underbrace{(D \to \mathbf{R})}_{\substack{\text{toplevel continuation}\\ \text{return}\\ \text{delimited continuation}}} \multimap \mathbf{R}) \multimap \underbrace{D \to \underbrace{(D \to \mathbf{R})}_{\substack{\text{toplevel continuation}\\ \text{call}\\ \text{delimited continuation}}} \multimap \mathbf{R}}$$

Since in Chapter 3 procedures were interpreted with continuation transformers (a type of continuations, or one isomorphic to a type of continuations), restricting all continuation types to linear usage cannot generally be sound, despite this interpretation's use of linearly-used delimited continuations.

On the other hand, we also gave an interpretation of first-class continuations:

$$D \stackrel{\text{def}}{=} \mu D.\ \underbrace{(D \to (D \to \mathbf{R}) \multimap \mathbf{R})}_{\substack{\text{return}\\ \text{delimited continuation}}} \to \underbrace{D \to (D \to \mathbf{R}) \multimap \mathbf{R}}_{\substack{\text{call}\\ \text{delimited continuation}}}$$

which uses undelimited continuations linearly, but delimited continuations nonlinearly. So restricting the use of too few continuation types (delimited continuations are isomorphic to a type of continuations) leads to a sound interpretation of `call/cc`, demonstrating that the linearity is effecting no restriction on control behavior whatsoever.

In summary: All the continuation types cannot be used linearly, and if too few are, things go horribly wrong. So instead, a particular semantics' representation of the control context must be identified, and then linear use analyzed.

Some, particularly those experienced with continuations or continuation semantics, may scoff at this point, thinking that of course one must choose the right notion of "continuation" for the situation at hand. While we agree, it remains that there is a semantic distinction between the notions of "destination of a jump" (which we term continuation) and "abstraction of the effect of the rest of the computation" (which we term control context). In some cases the two may coincide (for instance, they do for standard CPS semantics of the λ-calculus), but in general, control contexts are intimately tied to the language and particular semantics under consideration, while continuations can be defined once, irrespective of each particular language or semantics.

Chapter 6

A Simple Command Language

At this point we have presented the basic ideas of linearly used control contexts, but the line between which semantics and language features admit such interpretations and which do not is not particularly clear. We will now analyze the issues in more detail, sharpening the line. While we have used higher-order functional languages to present the basic ideas, continuing to do so would be highly inconvenient and artificial since we will need to explore languages with various binding and scoping mechanisms, various kinds of store, and various distinctions between different sorts of code. None of these fit well with extensions of λ-calculus, so we now shift to using a simple command language as the vehicle of our analysis.

Before considering any interesting control behavior, we briefly present an interpretation of a simple command language with virtually trivial control behavior. In following chapters we will extend this language in several independent directions, so it is convenient to present it on its own.

6.1 Source Syntax

The syntax of the source language is given by the grammar

$E ::=$		*expressions*
	$\mid n$	numeric literal
	$\mid *n$	dereference
	$\mid E$ `==` E	numeric equality
$C ::=$		*commands*
	$\mid n$ `=` E	assignment
	$\mid$ `skip`	no-op
	$\mid C$ `;` C	sequence
	$\mid$ `if(`E`){`C`}`	conditional

where n is a number. Here we are not concerned with issues of naming and binding and so forgo variables and instead work directly with locations, which we simply take to be numbers. For a simple example, the following command tests whether the contents of locations 1 and 2 are unequal, and if so swaps them using location 3 as a temporary:

$$\texttt{if(}*1 \texttt{ != } *2\texttt{)\{}3 \texttt{ = } *1\texttt{; }1 \texttt{ = } *2\texttt{; }2 \texttt{ = } *3\texttt{\}}$$

We give a standard direct semantics of this language shortly, but first we need to make some slight extensions to the target language.

6.2 Target Language $(+= \mathbf{N})$

Since numbers are central to the simple command language, we now admit that the target base type $\mathbf{N}$ is a type of numbers and flesh out the target language with numeric literals, numeric equality, and a test for zero. This is all straightforward.

6.2.1 Syntax

We extend the syntax with the productions

$$\begin{array}{lll} M ::= \cdots & & \textit{terms} \\ \quad \mid n & & \text{numeric literal} \\ \quad \mid M \stackrel{n}{=} M & & \text{numeric equality} \\ \quad \mid M \to M \parallel M & & \text{conditional (if zero)} \end{array}$$

Convention: Numeric equality has lower precedence than application, so $M\,N \stackrel{n}{=} O$ parses to $(M\,N) \stackrel{n}{=} O$ rather than $M\,(N \stackrel{n}{=} O)$. The conditional has lower precedence than numeric equality, so $M \stackrel{n}{=} N \to O \parallel P$ parses to $(M \stackrel{n}{=} N) \to O \parallel P$ rather than $M \stackrel{n}{=} (N \to O \parallel P)$.

For convenience, the numbers in the target language are taken to be the same numbers as in the source language.

6.2.2 Equational theory

We extend the equational theory with the following axioms:

$$\frac{}{n \stackrel{n}{=} n \stackrel{\beta\eta}{=} 1} \qquad \frac{}{m \stackrel{n}{=} n \stackrel{\beta\eta}{=} 0}\ m \neq n$$

$$\frac{}{0 \to M \parallel N \stackrel{\beta\eta}{=} M} \qquad \frac{}{n \to M \parallel N \stackrel{\beta\eta}{=} N}\ n \neq 0 \qquad \frac{}{n \to M \parallel M \stackrel{\beta\eta}{=} M}$$

Note that numeric equality encodes "true" and "false" as 1 and 0, and that the conditional tests for 0, not "true."

6.2.3 Type system

We extend the type system with the axiom and rules

$$[\textsc{NLit}]\ \frac{}{\Gamma\ ;\ -\ \vdash n : \mathbf{N}} \qquad [\textsc{NEq}]\ \frac{\Gamma\ ;\ \Delta \vdash M : \mathbf{N} \quad \Gamma\ ;\ \Delta' \vdash N : \mathbf{N}}{\Gamma\ ;\ \Delta\ ,\ \Delta' \vdash M \stackrel{n}{=} N : \mathbf{N}}$$

$$[\textsc{NCond}]\ \frac{\Gamma\ ;\ \Delta \vdash M : \mathbf{N} \quad \Gamma\ ;\ \Delta' \vdash N : A \quad \Gamma\ ;\ \Delta' \vdash O : A}{\Gamma\ ;\ \Delta\ ,\ \Delta' \vdash M \to N \parallel O : A}$$

Note how the two branches of the conditional share the same linear zone, just as the two factors of a $\&$-pair do.

6.3 Direct Semantics

Expressions are pure, so it is simplest to give a direct denotational semantics. The semantics of commands, on the other hand, is given by a big-step execution relation. In anticipation of relating the direct and continuation semantics, we use the target language as the metalanguage for defining the direct semantics.

6.3.1 States

Assignment in the source language necessitates the use of states. As usual, states map locations to values, which, in our case, are both numbers:

$$\mathbf{S} \stackrel{\text{def}}{=} \mathbf{N} \to \mathbf{N}$$

In Section A.1 we define some metalanguage syntactic sugar for manipulating states:

$$\begin{array}{lll} M ::= \cdots & & \textit{terms} \\ \quad \mid S[N] & & \text{lookup} \\ \quad \mid [S \mid N\!:\!M] & & \text{extension} \end{array}$$

Convention: State lookup has higher precedence than application, so $M\,S[N]$ parses to $M\,(S[N])$ rather than $(M\,S)[N]$.
And, in Section A.3 we derive typing rules for this sugar:

$$[\textsc{Lookup}]\ \frac{\Gamma\ ;\ \Delta \vdash S : \mathbf{N} \to A \quad \Gamma\ ;\ - \vdash N : \mathbf{N}}{\Gamma\ ;\ \Delta \vdash S[N] : A}$$

$$[\textsc{Extend}]\ \frac{\Gamma\ ;\ \Delta \vdash S : \mathbf{N} \to A \quad \Gamma\ ;\ - \vdash N : \mathbf{N} \quad \Gamma\ ;\ \Delta \vdash M : A}{\Gamma\ ;\ \Delta \vdash [S \mid N\!:\!M] : \mathbf{N} \to A}$$

These rules are more general than we need at this point, but in later chapters we will also use this sugar for (state-like) environments.

6.3.2 Expressions

Expressions have numeric value and may depend on the state, so they are interpreted with the type

$$\mathbf{S} \to \mathbf{N}$$

and the semantics is

$$\begin{array}{c} (\!|n|\!) \stackrel{\text{def}}{=} \lambda s.\, n \\ (\!|{*}n|\!) \stackrel{\text{def}}{=} \lambda s.\, s[n] \\ (\!|E_0 == E_1|\!) \stackrel{\text{def}}{=} \lambda s.\, (\!|E_0|\!)\, s \stackrel{n}{=} (\!|E_1|\!)\, s \end{array}$$

Proposition 13 (Soundness) *For any expression E*

$$-\ ;\ - \vdash (\!|E|\!) : \mathbf{S} \to \mathbf{N}$$

Proof By structural induction on the syntax of the expression. □

6.3.3 Commands

The semantics of commands is given by an execution relation, $(\cdot) \Downarrow (\cdot)$, which relates a command and a state to a state:

$$\frac{}{n = E, s \Downarrow [s \mid n\colon (\!|E|\!)\, s]} \qquad \frac{}{\texttt{skip}, s \Downarrow s} \qquad \frac{C_0, s \Downarrow s' \quad C_1, s' \Downarrow s''}{C_0\,;\, C_1, s \Downarrow s''}$$

$$\frac{C, s \Downarrow s'}{\texttt{if}(E)\ \{C\}, s \Downarrow s'}\ (\!|E|\!)\, s \overset{\beta\eta}{=} n \neq 0 \qquad \frac{}{\texttt{if}(E)\ \{C\}, s \Downarrow s}\ (\!|E|\!)\, s \overset{\beta\eta}{=} 0$$

6.4 Refined CPS Transformation

The interpretation we present is a refined version of, essentially, the continuation semantics given in [Rey98b]. The same interpretation of expressions as the direct semantics is used, but commands are given meaning by a CPS transformation.

In this simple language, once execution of a command finishes there is only a single way for the computation to proceed: to execute the lexically "next" command. So a command's control context is represented by a single command continuation, which abstracts the effect of the computation ensuing from execution of the next command. Commands simply execute, producing a new state without any other effects, and so the type of command continuations is

$$\mathbf{S} \to \mathbf{R}$$

Using this, commands are interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{current}\\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

which is a refined CPS version of the type of state transformers, $\mathbf{S} \to \mathbf{S}$. The transformation is shown in Figure 6.1.

Figure 6.1 Refined CPS of Simple Command Language

Expressions

$$\mathbf{S} \to \mathbf{N}$$

$$\begin{aligned}
(\!|n|\!) &\stackrel{\text{def}}{=} \lambda s.\, n \\
(\!|{*}n|\!) &\stackrel{\text{def}}{=} \lambda s.\, s[n] \\
(\!|E_0 == E_1|\!) &\stackrel{\text{def}}{=} \lambda s.\, (\!|E_0|\!)\, s \stackrel{n}{=} (\!|E_1|\!)\, s
\end{aligned}$$

Commands

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\text{current}} \multimap \mathbf{S} \to \mathbf{R}$$

$$\begin{aligned}
\overline{n = E} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, k\,[s \mid n\colon (\!|E|\!)\, s] \\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta k.\, k && \stackrel{\beta\eta}{=} \delta k.\, \lambda s.\, k\, s \\
\overline{C_0\,;\, C_1} &\stackrel{\text{def}}{=} \delta k.\, \overline{C_0}\llcorner(\overline{C_1}\llcorner k) && \stackrel{\beta\eta}{=} \delta k.\, \lambda s.\, \overline{C_0}\llcorner(\lambda s.\, \overline{C_1}\llcorner k\, s)\, s \\
\overline{\texttt{if}\,(E)\,\{C\}} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, (\!|E|\!)\, s \to k\, s \,[\!]\, \overline{C}\llcorner k\, s
\end{aligned}$$

6.5 Soundness

Since execution of a command can proceed in only one way, the soundness of this refined interpretation is similar to, but simpler than, that of interpretations in previous chapters.

Proposition 14 (Soundness) *For any command C*

$$- \,;\, - \vdash \overline{C} : (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}$$

Proof By structural induction on the syntax of the command. □

As before, technically the proof is basically self-evident. Conceptually, assignment is not problematic since the state is simply updated and passed to the current command continuation. `skip` is similar but the state is not modified. For sequencing, once the first command has been executed, there is no choice but to execute the second. After evaluating the test of a conditional command there is a choice of how to proceed, that is, there are two command continuations involved. Both of these continuations depend on the conditional command's current continuation, but this is not a problem since the branches of a target language conditional share the same restricted resources. This is very similar to how we dealt with the sharing needed between return and handler continuations using a &-pair in Chapter 4.

6.6 Adequacy

Since we make no distinction between $(\cdot) \to (\cdot)$ and $(\cdot) \multimap (\cdot)$ operationally, the refined CPS semantics behaves the same as the standard one. And since the big-step direct semantics of the source is standard, there are no surprises regarding computational adequacy.

Proposition 15 (Adequacy) *For any command C and state s, if*

$$C, s \Downarrow s'$$

then for any target term K

$$\overline{C}\llcorner K\, s \stackrel{\beta\eta}{=} K\, s'$$

Proof By structural induction on the derivation of the big-step judgment. □

6.7 Conclusion

Largely to set up upcoming languages and interpretations, we have seen how a simple command language with wholly uninteresting control behavior admits a refined interpretation using the type:

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{current} \\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

Chapter 7

Forward Jumps

A criticism of Chapter 4 is that there is only one sort of exceptions, and hence the flexibility of the possible control behavior the semantics must cope with is not what it ought to be. In this chapter we show that this in not an inherent limitation of linear use of control contexts by interpreting a language which allows many alternate jump destinations. Semantically, this corresponds to using a $\&$-tuple rather than $\&$-pair of continuations, and so is technically just a mild extension.

In the source language, the jump destinations are introduced by nested blocks binding labels which are jumped to with `goto`. This allows us to amend the situation of not having ventured outside the realm of structured programming with the source languages we have thus far interpreted with linear use of control contexts. There are two essentially independent aspects of this interpretation which we address separately in the next two chapters and later combine in Chapter 11. The first aspect is how nested blocks which bind labels are handled, which is the subject of this chapter. The second is how looping introduced by labeled jumping is handled, which is the subject of the next chapter.

Other than the "unstructured" nature of control flow introduced by `goto`, an additional point to note is that with the addition of labels, part of the control context is now *reified*, that is, named (by labels) in the source language [FW84]. However, since labels cannot be stored, they are not first-class, and so this point is not as significant as it might at first seem. That is, the ability to name continuations does not necessarily provide the ability to manipulate them as values, and so naming continuations does not necessarily grant the dramatic increase in expressiveness which first-class continuations provide. But the interpretation of labeled jumps which respects a discipline of linearly used control contexts does demonstrate that the connection between source identifiers and target identifiers need not be tight, and hence Contraction and Weakening of source identifiers (which may refer to pieces of control context) does not necessitate Contraction and Weakening of restricted target identifiers (which refer to control contexts).

7.1 Source Syntax

We now look at how to treat an environment of labels by extending the source language of Section 6.1 with labeled *forward* jumps. We call these jumps forward since they are to code which has not yet been executed, and hence no looping is introduced. Labels are a kind of name and so we introduce blocks to bind them and delimit their scope.

From a programmer's perspective, we would like blocks which bind multiple labels, $\{C_0\,;\, l_1 : C_1\,;\, \cdots\,;\, l_n : C_n\}$, but if we had single-label blocks, $\{C_0\,;\, l : C_1\}$, we could express multi-label blocks, $\{\{\cdots\{C_0\,;\, l_1 : C_1\}\,;\, \cdots\}\,;\, l_n : C_n\}$. Furthermore, since the second command in a single-label block has nothing to do with the scope of the label, we can avoid handling sequenc-

ing twice (both in sequence commands and in blocks) by eliminating sequencing from blocks, defining a block $\{C\ \ l\!:\!\}$ such that $\{C_0\ \ l\!:\!\}; C_1$ has the same intended meaning as $\{C_0; l\!:\! C_1\}$ would. So we extend the syntax of commands from Section 6.1 with the productions

$$\begin{array}{lll} C ::= \cdots & & \textit{commands} \\ \quad \mid \texttt{goto}\, l & & \text{jump} \\ \quad \mid \{C\ \ l\!:\!\} & & \text{block} \end{array}$$

where l is drawn from a set of labels.

As a simple example, although the conditional command in the simple command language has no "else branch" facility, the following command expresses the idiom using forward jumps:

$$\{\texttt{if}(E)\,\{C_{\text{then}}\,;\,\texttt{goto}\,l\};\, C_{\text{else}}\ \ l\!:\!\}$$

A block $\{C\ \ l\!:\!\}$ binds the label l in scope C. As usual, we work up to renaming of bound identifiers, which are labels here. Jumping to an unbound label is an error. Blocks are similar to a first-order binding form of `call/cc`. That is, $\{C\ \ l\!:\!\}$ is similar to `call/cc` $\lambda k.\, M$. One difference is that jumps to k carry arguments while those to l do not, but in the imperative language data can be passed through the store, so this difference is immaterial. Very significant, however, is that there is no way for l to escape C, while k could certainly escape from M, as demonstrated by `return/cc` in Section 3.6. This is a consequence of the fact that labels are not first-class while, with `call/cc`, continuations are. A result is that in the language of forward jumps, continuations are not upward, only downward. The implications the absence of upward continuations has on the refined interpretation are discussed below.

This language also provides the rudiments of a multiple exceptions mechanism since, with many labels there are many jump destinations, (possibly labeled) code between a `goto` command and the jump destination can be jumped over, and, nesting of blocks allows shadowing of outer bindings of a label. To properly capture the dynamic nature of exceptions, however, labels should be bound dynamically rather than statically, as they are here.[1] But since this language has no procedures, subroutines, etc., static and dynamic binding coincide.

7.2 Target Language $(+= \mathbf{1} + \&_n(\cdot))$

Before defining the semantics of the source language, some target language syntactic sugar for n-ary $\&$-tuples built from $\&$-pairs is convenient. And in order to do so, we must first introduce the unit type in the target language.

7.2.1 Syntax

We extend the grammar of terms of Section 2.2:

$$\begin{array}{lll} M ::= \cdots & & \textit{terms} \\ \quad \mid \langle\rangle & & \text{unit constant} \\ \quad \mid \lambda\langle\rangle.\, M & & \text{unit consumer} \\ \quad \mid \delta\langle\rangle.\, M & & \text{restricted unit consumer} \end{array}$$

And, in Section A.1 we define the following syntactic sugar:

[1] Later, in Chapter 11, when we combine this treatment with recursion and other features, we will address this concern and give a version where labels are bound dynamically.

$\mid \langle M \rangle$	unary $\&$-tuple	
$\mid \pi_i^n$	n-ary additive product projection	$0 \leq i < n$
$\mid \lambda\langle x, \ldots, x\rangle . M$	n-ary additive pattern match	
$\mid \delta\langle x, \ldots, x\rangle . M$	n-ary restricted additive pattern match	
$\mid [\langle M, \ldots, M\rangle \mid i{:}\, M]$	n-ary $\&$-tuple extension	

Convention: Unit consumers follow conventions similar to those for abstractions. When it is clear from context, we generally omit n and write π_i for π_i^n. Pattern match terms parse like abstractions.

7.2.2 Equational theory

We add the following to the axioms of Section 3.2:

$$\overline{(\lambda\langle\rangle . M)\ \langle\rangle \overset{\beta\eta}{=} M} \qquad\qquad \overline{(\delta\langle\rangle . M)_{\sqcup}\langle\rangle \overset{\beta\eta}{=} M}$$

And, in Section A.2 we define the following axiom for the syntactic sugar:

$$\overline{(\delta\langle x_0, \ldots, x_{n-1}\rangle . M)_{\sqcup}\langle M_0, \ldots, M_{n-1}\rangle \overset{\beta\eta}{=} M[x_0, \ldots, x_{n-1} \mapsto M_0, \ldots, M_{n-1}]}$$

7.2.3 Type system

The grammar of types of Section 3.2 is extended by

$P ::= \cdots$		*pointed types*
$\mid \mathbf{1}$		unit type

and, in Section A.3 we define the following syntactic sugar:

$\mid \&_n P$	n-ary additive product type

Convention: When it is clear from context, we generally omit n and write $\& P$ for $\&_n P$.

The typing rules of Section 3.2 are extended with

$$[\textsc{Unit}]\ \frac{}{\Gamma\ ;\ -\ \vdash\ \langle\rangle : \mathbf{1}}$$

$$[\textsc{AbsUnit}]\ \frac{\Gamma\ ;\ \Delta \vdash M : P}{\Gamma\ ;\ \Delta \vdash \lambda\langle\rangle . M : \mathbf{1} \to P} \qquad [\textsc{RAbsUnit}]\ \frac{\Gamma\ ;\ \Delta \vdash M : P}{\Gamma\ ;\ \Delta \vdash \delta\langle\rangle . M : \mathbf{1} \multimap P}$$

And, in Section A.3 we derive the following axioms for the syntactic sugar:

$$[\textsc{AProj}in]\ \frac{}{\Gamma\ ;\ -\ \vdash\ \pi_i^n : \&_n P \multimap P}$$

7.3 Standard CPS Transformation

So far we have given direct semantics of the various source languages under consideration, but a direct semantics for labeled jumps is hard to come by unless one works very concretely and essentially defines an abstract random access machine and compiles to it. But CPS is the most convenient abstraction of a random access machine which introduces very little intuitive or semantic gap, so we will present a standard continuation semantics of the source as well as a refined one.

This interpretation is essentially a reformulation of the semantics given by Strachey and Wadsworth [SW74], specialized to the more restricted source language. We use a direct semantics of expressions—that of Section 6.3.2—since they are *pure* (meaning that expressions cannot modify the state or alter control flow) in the language considered, which is not the case in Strachey and Wadsworth's language, and so they use a continuation semantics for expressions. It could be argued that the purity of expressions is artificial, but for our purposes here, little would be gained by not taking advantage of this simplification. And it could also be argued that at some level (operations on registers in a RISC machine, for instance), it makes perfect sense to consider pure expressions.[2]

Commands are interpreted with the type

$$\underbrace{\&\,(\mathbf{S}\to\mathbf{R})}_{\substack{\text{labeled continuations}\\ \text{(environment)}}} \quad\to\quad \underbrace{(\mathbf{S}\to\mathbf{R})}_{\substack{\text{current}\\ \text{continuation}}} \;\to \mathbf{S}\to\mathbf{R} \tag{7.1}$$

where in addition to the current continuation, there is one continuation for each label in scope. Note that while the type of this environment is a &-tuple of continuations, in the transformation we select continuations from this &-tuple by label. So essentially we have a mapping from labels to continuations, which looks much more like a standard environment. We could interpret commands more concisely by handling the environment implicitly, using the type

$$\underbrace{(\mathbf{S}\to\mathbf{R})}_{\substack{\text{current}\\ \text{continuation}}} \;\to \mathbf{S}\to\mathbf{R}$$

but the explicit environment is more true to [SW74] and closer to the refined interpretation we give later.

The transformation is parameterized by a nonrepetitive sequence $\vec{l}$ of the labels in scope:

$$\begin{aligned}
\overline{\overline{n = E}}_{\vec{l}} &\stackrel{\text{def}}{=} \lambda\langle\vec{l}\rangle.\,\lambda k.\,\lambda s.\,k\,[s \mid n\colon (\!|E|\!)\,s] \\
\overline{\overline{\texttt{skip}}}_{\vec{l}} &\stackrel{\text{def}}{=} \lambda\langle\vec{l}\rangle.\,\lambda k.\,k \\
\overline{\overline{C_0\,;\,C_1}}_{\vec{l}} &\stackrel{\text{def}}{=} \lambda\langle\vec{l}\rangle.\,\lambda k.\,\overline{\overline{C_0}}_{\vec{l}}\langle\vec{l}\rangle\,(\overline{\overline{C_1}}_{\vec{l}}\langle\vec{l}\rangle\,k) \\
\overline{\overline{\texttt{if}\,(E)\,\{C\}}}_{\vec{l}} &\stackrel{\text{def}}{=} \lambda\langle\vec{l}\rangle.\,\lambda k.\,\lambda s.\,(\!|E|\!)\,s \to k\,s\;[\!]\;\overline{\overline{C}}_{\vec{l}}\langle\vec{l}\rangle\,k\,s \\
\overline{\overline{\texttt{goto}\,l}}_{\vec{l}} &\stackrel{\text{def}}{=} \lambda\langle\vec{l}\rangle.\,\lambda k.\,l \\
\overline{\overline{\{C\;\;l\colon\}}}_{\vec{l}} &\stackrel{\text{def}}{=} \lambda\langle\vec{l}\rangle.\,\lambda k.\,\overline{\overline{C}}_{\vec{l},l}\,\langle\vec{l},k\rangle\,k \qquad\qquad l\notin\vec{l}
\end{aligned}$$

Computationally, the first four clauses are very similar to the transformation in Section 6.4. The only difference is that the label environment must be passed along like the current continuation, but these clauses do not manipulate any of the labeled continuations. The transformation

[2] Additionally, in Chapter 11, we use a continuation semantics of expressions.

of a block adds the label to the sequence of labels in scope for the body command, and passes the current continuation as both the newly labeled continuation and the current continuation. The transformation of `goto` is analogous to that of `skip`, except that the continuation associated with the label is invoked instead of the current continuation.

Note that since we identify commands up to renaming of bound labels, the side condition of the clause for blocks can always be satisfied. Also, jumping to an unbound label, an error in the source language, results in an open target term. It is necessary to parameterize the transformation because the transformation of a command must abstract over the labels in scope, but which labels have scope including the command being transformed cannot be determined from the command alone. An implicit environment interpretation would not require this added complexity but would instead rely on substitution.

Proposition 16 (Soundness) *For any command C, if the nonrepetitive sequence of labels $\vec{l} = l_1, \ldots, l_n$ (for some $n \geq 0$) contains the free labels of C, then*

$$l_1 : \mathbf{S} \to \mathbf{R}\,, \ldots, l_n : \mathbf{S} \to \mathbf{R}\,;\, - \vdash \overline{\overline{C}}_{\vec{l}} : \&_n\,(\mathbf{S} \to \mathbf{R}) \to (\mathbf{S} \to \mathbf{R}) \to \mathbf{S} \to \mathbf{R}$$

Proof By structural induction on the syntax of the command. □

7.4 Refined CPS Transformation

A command's control context consists of the labeled continuations and the current continuation. A first crack at a version of (7.1) which uses the control context linearly could be

$$\underbrace{\&\,(\mathbf{S} \to \mathbf{R})}_{\substack{\text{labeled}\\ \text{continuations}}} \multimap \underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{current}\\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

but this does not work because it treats the current and labeled continuations differently, while in the source language they are invoked in essentially the same way. This can be seen by noting that "`skip`" means "`goto` the current continuation," and hence the fact that the current continuation is not named while the labeled continuations are does not imply that they are manipulated differently. So, similar to Section 4.3, we treat accepting the labeled and current continuation arguments symmetrically by combining them all in a $\&$-tuple and interpret commands with the type

$$\underbrace{\&\,(\mathbf{S} \to \mathbf{R})}_{\substack{\text{labeled}\\ \text{continuations}}} \;\&\; \underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{current}\\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

Notice that handling the labeled and current continuations symmetrically has eliminated the option of handling the label environment implicitly, since doing so would require handling the current continuation implicitly, which is problematic because an implicit environment depends on substitution, while the current continuation is unnamed and effectively bound dynamically.

The transformation does not differ from the standard one computationally, the differences are all a result of the change in the type used to interpret commands. As before, the transformation is parameterized by a nonrepetitive sequence $\vec{l}$ of the labels in scope, see Figure 7.1. Again, since we identify commands up to renaming of bound labels, the side condition of the clause for blocks can always be satisfied.

Figure 7.1 Refined CPS of Forward Jumps

Expressions

$$\mathbf{S} \to \mathbf{N}$$

$$\begin{aligned}
(\!|n|\!) &\stackrel{\text{def}}{=} \lambda s.\, n \\
(\!|{*}n|\!) &\stackrel{\text{def}}{=} \lambda s.\, s[n] \\
(\!|E_0 == E_1|\!) &\stackrel{\text{def}}{=} \lambda s.\, (\!|E_0|\!)\, s \stackrel{n}{=} (\!|E_1|\!)\, s
\end{aligned}$$

Commands

$$\underbrace{\&\,(\mathbf{S} \to \mathbf{R})}_{\text{labels}} \,\&\, \underbrace{(\mathbf{S} \to \mathbf{R})}_{\text{current}} \multimap \mathbf{S} \to \mathbf{R}$$

$$\begin{aligned}
\overline{n = E}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle \vec{l}, k\rangle.\, \lambda s.\, k\,[s \mid n\colon (\!|E|\!)\, s] \\
\overline{\texttt{skip}}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle \vec{l}, k\rangle.\, k \\
\overline{C_0\,;\, C_1}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle \vec{l}, k\rangle.\, \overline{C_0}_{\vec{l}}\langle \vec{l}, \overline{C_1}_{\vec{l}}\langle \vec{l}, k\rangle\rangle \\
\overline{\texttt{if(}E\texttt{)}\ \{C\}}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle \vec{l}, k\rangle.\, \lambda s.\, (\!|E|\!)\, s \to k\, s \,[]\, \overline{C}_{\vec{l}}\langle \vec{l}, k\rangle\, s \\
\overline{\texttt{goto}\ l}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle \vec{l}, k\rangle.\, l \\
\overline{\{C\ \ l\colon\}}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle \vec{l}, k\rangle.\, \overline{C}_{\vec{l},l}\langle \vec{l}, k, k\rangle \qquad\qquad l \notin \vec{l}
\end{aligned}$$

7.5 Soundness

The typing techniques employed by the interpretation in the previous section are very similar to those used in Chapter 4. The common problem which using &-product addresses is allowing construction and invocation of multiple continuations which share a continuation. In particular, notice how the two occurrences of k in the transformation of a block now occur within a &-tuple, allowing linear typechecking. Similarly for the two occurrences of $\vec{l}$ in the transformation of a sequence. Also, the transformations of `skip` and `goto` do not require Weakening since one of the factors of the &-tuple is kept.

Earlier, we discussed how continuations are only used in a downward, not upward, fashion in this interpretation. This characteristic is crucial to the soundness of this interpretation. First, notice how the transformation manipulates the &-tuple of labeled and current continuations in an environment-like, rather than a state-like, fashion. That is, the &-tuple is passed "down" to each of a command's subcommands, but never returned back "up" by passing it to a continuation. The &-tuple may be extended, but the extension is only active for the command under consideration; in other words, &-tuple extension introduces a local binding rather than performs an assignment. Also, whenever an individual continuation is projected from the &-tuple, the rest of the &-tuple is no longer needed since other subcommands will have been passed their own &-tuple. This is significant because if the source language required upward continuations, a &-tuple would be needed after a continuation had been projected from it, which would break linearity. This does not mean that control constructs which require upward continuations cannot be interpreted with linear use of control contexts, but a different attack is needed, which we demonstrate in Chapter 9 where we give an interpretation of coroutines.

Another possibility is to have continuations which are upward not because they are used as pieces of state (as in the interpretation of coroutines, as we will see) but because they are expressed values (which would occur if we allowed labels as expressions). Semantically, these

two situations are much the same since either as expressed values or as part of a control store, upward continuations are arguments to other continuations. We explore these possibilities in Chapter 10.

Proposition 17 (Soundness) *For any command C, if the nonrepetitive sequence of labels* $\vec{l} = l_1, \ldots, l_n$ *(for some* $n \geq 0$*) contains the free labels of C, then*

$$- \,;\, - \vdash \overline{C}_{\vec{l}} : \mathop{\&}_n (\mathbf{S} \to \mathbf{R}) \,\&\, (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}$$

Proof By structural induction on the syntax of the command. □

7.6 Adequacy

The difference between the standard and refined interpretations is some uncurrying and restricting an $(\cdot) \to (\cdot)$ to a $(\cdot) \multimap (\cdot)$. Neither of these changes is computationally significant and so adequacy is unproblematic.

Proposition 18 (Adequacy) *For any command C, term K, terms* $\vec{K} = K_1, \ldots, K_n$*, and nonrepetitive sequence of labels* $\vec{l} = l_1, \ldots, l_n$ *(for some* $n \geq 0$*)*

$$\overline{C}_{\vec{l}\llcorner}\langle \vec{K}, K \rangle \stackrel{\beta\eta}{=} \overline{\overline{C}}_{\vec{l}} \langle \vec{K} \rangle \, K$$

Proof By structural induction on the syntax of the command. □

7.7 Conclusion

In this chapter we have seen how to a language with forward jumps, to code not already executed. This restriction allows us to dodge the complication of loops (until next chapter) while investigating the treatment of nested scopes binding continuations. The interpretation:

$$\underbrace{\mathop{\&} (\mathbf{S} \to \mathbf{R})}_{\substack{\text{labeled} \\ \text{continuations}}} \,\&\, \underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{current} \\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

uses a $\&$-tuple of continuations to represent the control environment. The primary point is that, for such a language, a semantics with only an environment suffices, and the continuations are passed only "downward."

Since we already knew how to treat exceptions using a $\&$-pair, technically this is only a slight extension.

Chapter 8

Backward Jumps

In Chapter 7 we considered some issues of label scope and binding, and the basic mechanics of jumping. The other primary aspect of interpreting `goto` is handling the possibility of jumping backwards—to previously executed code—which introduces the potential for looping. We came across this issue in the interpretation of untyped λ-calculus in Chapter 3 and dodged the difficulties since the standard interpretation did not rely on recursive continuations; that is, recursion was present at the level of continuation transformers rather than continuations. The setting of this chapter, however, is not so accommodating: the standard interpretation as introduced by Strachey and Wadsworth [SW74] does make use of recursive continuations, and the gentlest refinement breaks linearity. This forces us to use an interpretation which uses recursive delimited continuations (or continuation transformers) instead of recursive continuations.

8.1 Source Language

In this chapter we will only be concerned with issues of looping, not binding and scoping of labels which was handled in Chapter 7. So we extend the source language of Section 6.1, in an independent direction from the extension of Section 7.1, with programs which bind a label whose scope is the entire program.

8.1.1 Syntax

We extend the syntax of Section 6.1 with the productions

$$\begin{array}{rll} C ::= & \cdots & \textit{commands} \\ & \mid \texttt{goto l} & \text{jump} \\ P ::= & \texttt{l:}\, C & \textit{programs} \end{array}$$

where `l` is the single label.

To get a feel for the intended semantics, note that executing[1]

$$\texttt{l: if(*0 != 0) \{0 = *0 - 1; goto l\}}$$

decrements the contents of location 0 until it reaches 0, or, if 0 contained a negative number to start with, diverges.

[1] Though we do not treat them formally, in this example we make use of numeric inequality and subtraction primitives.

8.1.2 Standard CPS transformation

As for the language of forward jumps in Chapter 7, a direct semantics of labeled jumping is inconvenient and we instead give a standard continuation semantics of the source. As in Section 7.3, this interpretation is essentially a reformulation of the semantics given by Strachey and Wadsworth [SW74], specialized to the more restricted source language. As before, we use a direct semantics of expressions and handle the environment explicitly. Also, Strachey and Wadsworth handle recursion implicitly, while we handle it explicitly.

Other than the absence of blocks and the fact that there is only a single label in the environment, the transformation is the same as that in Section 7.3. Commands are interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{labeled}\\ \text{continuation}}} \to \underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{current}\\ \text{continuation}}} \to \mathbf{S} \to \mathbf{R}$$

and the transformation is

$$\begin{aligned}
\overline{\overline{n = E}} &\stackrel{\text{def}}{=} \lambda l.\, \lambda k.\, \lambda s.\, k\,[s \mid n\colon (\!|E|\!)\, s] \\
\overline{\overline{\texttt{skip}}} &\stackrel{\text{def}}{=} \lambda l.\, \lambda k.\, k \\
\overline{\overline{C_0\,;\, C_1}} &\stackrel{\text{def}}{=} \lambda l.\, \lambda k.\, \overline{C_0}\, l\, (\overline{C_1}\, l\, k) \\
\overline{\overline{\texttt{if}(E)\,\{C\}}} &\stackrel{\text{def}}{=} \lambda l.\, \lambda k.\, \lambda s.\, (\!|E|\!)\, s \to k\, s \,[\!]\, \overline{C}\, l\, k\, s \\
\overline{\overline{\texttt{goto l}}} &\stackrel{\text{def}}{=} \lambda l.\, \lambda k.\, l
\end{aligned}$$

Instead of blocks, the source language now includes programs, which are interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \to \mathbf{S} \to \mathbf{R}$$

and the transformation is

$$\overline{\overline{\texttt{l}\colon C}} \stackrel{\text{def}}{=} \lambda t.\, \mathsf{Y}\, \lambda l.\, \overline{C}\, l\, t \tag{8.1}$$

where

$$\begin{aligned}
\mathsf{Y} &\stackrel{\text{def}}{=} (\lambda z.\, \lambda x.\, x\,(z\,z\,x))\, \lambda z.\, \lambda x.\, x\,(z\,z\,x) : (P \to P) \to P \\
&\stackrel{\beta\eta}{=} \lambda x.\, x\,(\mathsf{Y}\,x)
\end{aligned}$$

is a standard Church-style call-by-name least fixed-point combinator (see Lemma 35). Note that Y is a self-application of a term of type $\mu Z.\, Z \to (P \to P) \to P$, but since we use equirecursive types in the target language, we can use the same term as in untyped λ-calculus. (Instead of defining Y to be a particular λ-term, we could just add a constant to the target language corresponding to the least fixed-point finder in the denotational model. But, since we will later need to consider nonstandard fixed-point combinators for refined typing purposes, it is clearer to use explicit λ-terms. Also, since we will need fixed-point combinators at nonstandard types, explicitly defining them as λ-terms provides much greater confidence in their existence, which is otherwise not immediately clear.)

Here we see that a program accepts its toplevel continuation and produces a continuation which is obtained by recursively passing itself, and then passing the toplevel continuation, to the body. So in the body of the program, the label is bound to the continuation representing the whole program.

Proposition 19 (Soundness) *1. For any command C*

$$- \,;\, - \vdash \overline{\overline{C}} : (\mathbf{S} \to \mathbf{R}) \to (\mathbf{S} \to \mathbf{R}) \to \mathbf{S} \to \mathbf{R}$$

2. For any program P

$$- \,;\, - \vdash \overline{\overline{P}} : (\mathbf{S} \to \mathbf{R}) \to \mathbf{S} \to \mathbf{R}$$

Proof

1. By structural induction on the syntax of the command.

2. Follows from 1. □

8.2 Refined CPS Transformation

8.2.1 Commands

The refined interpretation of commands is merely a simplification of the interpretation of Section 7.4 to handle only a single label. Hence commands are now interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{labeled}\\ \text{continuation}}} \;\&\; \underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{current}\\ \text{continuation}}} \;\multimap \mathbf{S} \to \mathbf{R}$$

and the transformation is shown in Figure 8.1.

8.2.2 Programs

As is probably expected, programs are interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \;\multimap \mathbf{S} \to \mathbf{R} \tag{8.2}$$

First attempt: unsound

An attempt at a refined analogue of the standard transformation is

$$\overline{\overline{\mathtt{l}:C}} = \delta t.\, \mathsf{Y}^{\circ}\, \delta l.\, \overline{C} \llcorner \langle l, t \rangle \tag{8.3}$$

(where Y°, which we define later, is a least fixed-point combinator of type $(P \multimap P) \to P$). But, although the dynamic behavior of this interpretation is linear in its use of t, as can be more easily seen by rewriting (8.3)

$$\overline{\overline{\mathtt{l}:C}} \llcorner t = \overline{C} \llcorner \langle \overline{\overline{\mathtt{l}:C}} \llcorner t, t \rangle$$

the free continuation, t, in the argument to Y° in (8.3) forces linear typechecking to fail.

Second attempt: sound (and adequate), but wrong definition of control context
We can work around this problem by moving up a level in the types to close the argument to Y:

$$\overline{\texttt{1}:C} = \mathsf{Y}\,\lambda x.\,\delta t.\,\overline{C}\llcorner\langle x \llcorner t, t\rangle \tag{8.4}$$

The difference between (8.4) and (8.3) is that in (8.3) the toplevel is accepted and immediately buried in a recursive continuation, while in (8.4) we form a recursive delimited continuation. Here, unlike in Chapter 5, it is more convenient to represent delimited continuations with the type

$$(\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}$$

rather than with the type

$$\mathbf{S} \to (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{R}$$

The recursive delimited continuation formed in (8.4) does not contain any continuation, but instead accepts the toplevel continuation and subsequently passes it on to each recursive call. This avoids creating a closure with a piece of control context inside, which the restricted type system is not happy with.

A result of moving up a level in the types is that different backward jumps correspond to *distinct* continuations, which are generated by fixed-point unwinding. So although there may be many jumps to the label, each continuation is only invoked once. To make this concrete, it is helpful to consider an example of the effect of unwinding. Explicitly, unwinding (8.4) twice we have

$$\overline{\texttt{1}:C} = \delta t.\,\overline{C}\llcorner\langle\underbrace{\overline{C}\llcorner\langle\overbrace{(\mathsf{Y}\,\lambda x.\,\delta t.\,\overline{C}\llcorner\langle x \llcorner t, t\rangle)\llcorner t}^{\text{second}}, t\rangle}_{\text{first}}, t\rangle$$

From this we see that jumps from the first occurrence of C will invoke the continuation labeled "first," while jumps from the second occurrence of C will invoke the continuation labeled "second."

This treatment of recursion is very similar to the handling of recursion in untyped λ-calculus in Chapter 3 where continuation transformers are self-applied to unwind to a fixed-point, but continuations are not recursive. The difference here is that we explicitly take a fixed-point in the transformation, rather than rely on self-application in the source language programs. While (8.4) typechecks with the right type at toplevel:

Proposition 20 (Soundness) *For any program P*

$$- \,;\, - \vdash \overline{P} : (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}$$

if we consider the types of the subterms of (8.4), we see that this interpretation—like the interpretation of first-class continuations in Section 5.4—restricts the use of part of the control context (the toplevel continuation), but does not restrict the use of another part (the recursive delimited continuation), making the meaningfulness of Proposition 20 dubious at best.[2]

It is obvious that (8.4) does not restrict the use of the recursive delimited continuation, x, but it is not as apparent that x is actually part of the control context. It helps to consider what happens at the level of types by thinking of "moving up a level in the types" as "partially specifying the result type." In (8.4), the type of $\lambda x.\,\delta t.\,\overline{C}\llcorner\langle x \llcorner t, t\rangle$ is

$$((\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}) \to (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}$$

[2]This point was overlooked in the previous presentation of this work [BORT02].

Changing from continuation-first to continuation-second form, this is isomorphic to

$$(\mathbf{S} \to (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{R}) \to \mathbf{S} \to (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{R}$$

or equivalently

$$(\mathbf{S} \to \mathbf{R}') \to \mathbf{S} \to \mathbf{R}' \quad \text{where} \quad \mathbf{R}' \stackrel{\text{def}}{=} (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{R} \tag{8.5}$$

This is similar to the type

$$(\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R} \tag{8.6}$$

which we attempted to give $\delta l.\, \overline{C}_{\sqcup}\langle l, t\rangle$ in (8.3), but with a more specified result type.

But (8.5) is not simply (8.6) with a more refined result type since use of the argument continuation is unrestricted in (8.5). So, "moving up a level in the types" from (8.3) to (8.4) corresponds not only to "partially specifying the result type" from $\mathbf{R}$ to $(\mathbf{S} \to \mathbf{R}) \multimap \mathbf{R}$, which is harmless, but also to changing the principal $(\cdot) \multimap (\cdot)$ to $(\cdot) \to (\cdot)$, which relaxes the usage restrictions on the control context. Hence, instead of (8.5), we need to use the type

$$(\mathbf{S} \to \mathbf{R}') \multimap \mathbf{S} \to \mathbf{R}' \tag{8.7}$$

Third attempt: sanity restored
In order to define an interpretation using type (8.7), we need a term of type

$$((\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}) \multimap (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}$$

to fill the role of $\lambda x.\, \delta t.\, \overline{C}_{\sqcup}\langle x_{\sqcup} t, t\rangle$ in (8.4). We do this with the transformation

$$\overline{\mathtt{1} : C} \stackrel{\text{def}}{=} \mathsf{Y}^{\circ}\, \delta x.\, \delta t.\, \overline{C}_{\sqcup}\langle x_{\sqcup} t, t\rangle \tag{8.8}$$

where

$$\begin{aligned} \mathsf{Y}^{\circ} &\stackrel{\text{def}}{=} \mathsf{Y}\, \lambda y.\, \lambda x.\, x_{\sqcup}(y\, x) : (P \multimap P) \to P \\ &\stackrel{\beta\eta}{=} \lambda x.\, x_{\sqcup}(\mathsf{Y}^{\circ}\, x) \end{aligned}$$

is a least fixed-point combinator:

Lemma 21 (Y° computes least fixed-points)

$$[\![- \,;\, - \vdash \mathsf{Y}^{\circ} : (P \multimap P) \to P]\!]\, [\,]\, [\,] = [\![- \,;\, - \vdash \mathsf{Y} : (P \to P) \to P]\!]\, [\,]\, [\,]$$

Here we use the standard predomain semantics of the target language, which is briefly described in Section A.4.

Proof Straightforward calculation depending on Lemma 34. □

Note that (8.8) still does not typecheck linearly since x is thrown away in the right factor of the &-pair. To typecheck this requires a Weakening rule for the restricted zone:

$$[\text{RWEAK}]\ \frac{\Gamma\ ;\ \Delta \vdash M : B}{\Gamma\ ;\ \Delta\ ,\ x : A \vdash M : B}$$

When the target type system is augmented with [RWEAK] it becomes an affine, rather than linear, type system. We are happy to add Weakening since we cannot see a conceptual reason to disallow it, and its addition does not invalidate any other results we show, including no junk. However, Weakening can be hacked around in this case and so in Section 8.5 we present a purely linear interpretation. Whether this supports or undermines the case for allowing Weakening is debatable.

Figure 8.1 Refined CPS of Backward Jumps

Expressions

$$\mathbf{S} \to \mathbf{N}$$

$$\begin{aligned}
(\!|n|\!) &\stackrel{\text{def}}{=} \lambda s.\, n \\
(\!|{*}n|\!) &\stackrel{\text{def}}{=} \lambda s.\, s[n] \\
(\!|E_0 \texttt{ == } E_1|\!) &\stackrel{\text{def}}{=} \lambda s.\, (\!|E_0|\!)\, s \stackrel{n}{=} (\!|E_1|\!)\, s
\end{aligned}$$

Commands

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\text{label}} \,\&\, \underbrace{(\mathbf{S} \to \mathbf{R})}_{\text{current}} \multimap \mathbf{S} \to \mathbf{R}$$

$$\begin{aligned}
\overline{n = E} &\stackrel{\text{def}}{=} \delta\langle l, k\rangle.\, \lambda s.\, k\,[s \mid n\colon (\!|E|\!)\, s] \\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta\langle l, k\rangle.\, k \\
\overline{C_0\,;\, C_1} &\stackrel{\text{def}}{=} \delta\langle l, k\rangle.\, \overline{C_0}\,_\langle l, \overline{C_1}\,_\langle l, k\rangle\rangle \\
\overline{\texttt{if(}E\texttt{)\{}C\texttt{\}}} &\stackrel{\text{def}}{=} \delta\langle l, k\rangle.\, \lambda s.\, (\!|E|\!)\, s \to k\, s \,[]\, \overline{C}\,_\langle l, k\rangle\, s \\
\overline{\texttt{goto l}} &\stackrel{\text{def}}{=} \delta\langle l, k\rangle.\, l
\end{aligned}$$

Programs

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{S} \to \mathbf{R}$$

$$\overline{\texttt{l}\colon C} \stackrel{\text{def}}{=} \mathsf{Y}^{\circ}\, \delta x.\, \delta t.\, \overline{C}\,\_\langle x\_t, t\rangle$$

8.3 Soundness

There is nothing technically remarkable regarding soundness.

Proposition 22 (Soundness) *1. For any command C*

$$- \,;\, - \vdash \overline{C} : (\mathbf{S} \to \mathbf{R}) \,\&\, (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}$$

2. For any program P

$$- \,;\, - \vdash \overline{P} : (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}$$

Proof

1. By structural induction on the syntax of the command.
2. Follows from 1. □

8.4 Adequacy

As for forward jumps, the differences between the standard and refined interpretations of commands are not computationally significant, and so adequacy is straightforward.

Proposition 23 (Adequacy for Commands) *For any command C and terms L, K*

$$\overline{C}\llcorner\langle L, K\rangle \stackrel{\beta\eta}{=} \overline{\overline{C}}\, L\, K$$

Proof By structural induction on the syntax of the command. □

For programs, adequacy is more subtle. In fact $\beta\eta$-equality is too fine an equivalence to allow us to prove the correspondence between the two transformations. This is expected since we are trying to establish the equivalence of two possibly-divergent terms. So, to prove adequacy we appeal to the standard predomain semantics of the target language given in Section A.4.

Pictorially, we have the following situation:

$$\begin{array}{c} \text{source} \\ \text{language} \end{array} \underset{\overline{(\cdot)}}{\overset{\overline{\overline{(\cdot)}}}{\rightleftarrows}} \begin{array}{c} \text{target} \\ \text{language} \end{array} \xrightarrow{[\![(\cdot)]\!]} \text{predomains}$$

Strachey and Wadsworth's original semantics is essentially the composition of the predomain semantics and the standard CPS transformation, $[\![\overline{\overline{(\cdot)}}]\!]$. We prove the adequacy of the refined CPS transformation by showing that its composition with the predomain semantics, $[\![\overline{(\cdot)}]\!]$, is equal to $[\![\overline{\overline{(\cdot)}}]\!]$. Any two source language programs are equivalent if $[\![\overline{\overline{(\cdot)}}]\!]$ maps them to equal predomain elements, so if programs are mapped to equal predomain elements by $[\![\overline{(\cdot)}]\!]$ and $[\![\overline{\overline{(\cdot)}}]\!]$, then source programs which are mapped to equal predomain elements by $[\![\overline{(\cdot)}]\!]$ are equivalent.

Proposition 24 (Adequacy for Programs) *For any program P, terms S, K such that*

$$- \,;\, t : \mathbf{S} \to \mathbf{R} \vdash K : \mathbf{S} \to \mathbf{R} \qquad\qquad - \,;\, - \vdash S : \mathbf{S}$$

we have

$$[\![- \,;\, - \vdash \overline{P}\llcorner K[t \mapsto \lambda s.\,\mathsf{halt}]\, S : \mathbf{R}]\!]\,[\,]\,[\,] = [\![- \,;\, - \vdash \overline{\overline{P}}\, K[t \mapsto \lambda s.\,\mathsf{halt}]\, S : \mathbf{R}]\!]\,[\,]\,[\,]$$

In order to avoid comparing the semantics of terms of different types, we compare the results of running the program under both semantics rather than the programs themselves. To do this, we have compromised the abstractness of $\mathbf{R}$ by introducing a closed term of result type,[3] with syntax

$$\begin{array}{rll} M ::= & \cdots & \textit{terms} \\ & \mid \mathsf{halt} & \text{terminate} \end{array}$$

and typing

$$[\textsc{Halt}]\ \frac{}{\Gamma \,;\, - \vdash \mathsf{halt} : \mathbf{R}}$$

Intuitively, halt represents all the actions the operating system performs when a program relinquishes control to it. Having this facility in the programming language is clearly not very sensible, but the simple but inaccurate standard model suffices for our purposes here, and a proper account of the abstractness of the result type has not yet been given.

[3] Actually, the standard predomain semantics already violated the abstractness of $\mathbf{R}$ by interpreting it with a particular domain. So by introducing halt we are simply pulling this from the semantics into the language.

Proof Fix $P = \mathtt{1}\colon C$, and S, K such that $- \,;\, t : \mathbf{S} \to \mathbf{R} \vdash K : \mathbf{S} \to \mathbf{R}$ and $- \,;\, - \vdash S : \mathbf{S}$. Then we have

$$
\begin{aligned}
&[\![- \,;\, - \vdash \overline{P} \lrcorner K[t \mapsto \lambda s.\, \mathsf{halt}]\, S : \mathbf{R}]\!]\, [\,]\, [\,] \\
&= [\![- \,;\, - \vdash \overline{\mathtt{1}\colon C} \lrcorner K[t \mapsto \lambda s.\, \mathsf{halt}]\, S : \mathbf{R}]\!]\, [\,]\, [\,] \\
&= [\![- \,;\, - \vdash (\mathsf{Y}^\circ\, \delta x.\, \delta t.\, \overline{C} \llcorner \langle x \lrcorner t, t\rangle) \lrcorner K[t \mapsto \lambda s.\, \mathsf{halt}]\, S : \mathbf{R}]\!]\, [\,]\, [\,] \\
&= [\![- \,;\, - \vdash (\mathsf{Y}^\circ\, \delta x.\, \delta t.\, \overline{C} \llcorner \langle x \lrcorner t, t\rangle) \lrcorner K[t \mapsto \lambda s.\, \mathsf{halt}] : \neg\mathbf{S}]\!]\, [\,]\, [\,]\, ([\![- \,;\, - \vdash S : \mathbf{S}]\!]\, [\,]\, [\,]) \\
&= [\![- \,;\, - \vdash \mathsf{Y}^\circ\, \delta x.\, \delta t.\, \overline{C} \llcorner \langle x \lrcorner t, t\rangle : \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\, [\,]\, [\,]\, ([\![- \,;\, - \vdash K[t \mapsto \lambda s.\, \mathsf{halt}] : \neg\mathbf{S}]\!]\, [\,]\, [\,]) \\
&\quad ([\![- \,;\, - \vdash S : \mathbf{S}]\!]\, [\,]\, [\,])
\end{aligned}
$$

and

$$
\begin{aligned}
&[\![t : \neg\mathbf{S} \,;\, - \vdash \mathsf{Y}^\circ\, \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S}]\!]\, [\,]\, [[\,] \mid t\colon [\![- \,;\, - \vdash K[t \mapsto \lambda s.\, \mathsf{halt}] : \neg\mathbf{S}]\!]\, [\,]\, [\,]] \\
&\quad ([\![- \,;\, - \vdash S : \mathbf{S}]\!]\, [\,]\, [\,]) \\
&= (\lambda d \in [\![- \vdash \neg\mathbf{S}]\!]\, [\,].\, [\![t : \neg\mathbf{S} \,;\, - \vdash \mathsf{Y}^\circ\, \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S}]\!]\, [\,]\, [[\,] \mid t\colon d]) \\
&\quad ([\![- \,;\, - \vdash K[t \mapsto \lambda s.\, \mathsf{halt}] : \neg\mathbf{S}]\!]\, [\,]\, [\,])\, ([\![- \,;\, - \vdash S : \mathbf{S}]\!]\, [\,]\, [\,]) \\
&= [\![- \,;\, - \vdash \lambda t.\, \mathsf{Y}^\circ\, \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S} \to \neg\mathbf{S}]\!]\, [\,]\, [\,] \\
&\quad ([\![- \,;\, - \vdash K[t \mapsto \lambda s.\, \mathsf{halt}] : \neg\mathbf{S}]\!]\, [\,]\, [\,])\, ([\![- \,;\, - \vdash S : \mathbf{S}]\!]\, [\,]\, [\,]) \\
&= [\![- \,;\, - \vdash (\lambda t.\, \mathsf{Y}^\circ\, \lambda l.\, \overline{\overline{C}}\, l\, t)\, K[t \mapsto \lambda s.\, \mathsf{halt}] : \neg\mathbf{S}]\!]\, [\,]\, [\,]\, ([\![- \,;\, - \vdash S : \mathbf{S}]\!]\, [\,]\, [\,]) \\
&= [\![- \,;\, - \vdash (\lambda t.\, \mathsf{Y}^\circ\, \lambda l.\, \overline{\overline{C}}\, l\, t)\, K[t \mapsto \lambda s.\, \mathsf{halt}]\, S : \mathbf{R}]\!]\, [\,]\, [\,] \\
&= [\![- \,;\, - \vdash \overline{\overline{P}}\, K[t \mapsto \lambda s.\, \mathsf{halt}]\, S : \mathbf{R}]\!]\, [\,]\, [\,]
\end{aligned}
$$

So, let $k = [\![- \,;\, - \vdash K[t \mapsto \lambda s.\, \mathsf{halt}] : \neg\mathbf{S}]\!]\, [\,]\, [\,]$, and the result follows from

$$[\![- \,;\, - \vdash \mathsf{Y}^\circ\, \delta x.\, \delta t.\, \overline{C} \llcorner \langle x \lrcorner t, t\rangle : \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\, [\,]\, [\,]\, k = [\![t : \neg\mathbf{S} \,;\, - \vdash \mathsf{Y}^\circ\, \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S}]\!]\, [\,]\, [[\,] \mid t\colon k]$$

Now we have

$$\begin{aligned}
&[\![- \,;\, - \vdash \mathsf{Y}^\circ\, \delta x.\, \delta t.\, \overline{C}_{\llcorner}\langle x_{\llcorner} t, t\rangle : \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\, [\,]\, [\,]\, k \\
&= \bigsqcup\{([\![- \,;\, - \vdash \delta x.\, \delta t.\, \overline{C}_{\llcorner}\langle x_{\llcorner} t, t\rangle : (\neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\, [\,]\, [\,])^n \bot \mid n \geq 0\}\, k \\
&\quad \text{by Lemma 35} \\
&= (\lambda f \in [\![- \vdash \neg\mathbf{S} \multimap \neg\mathbf{S}]\!].\, f\, k) \\
&\quad \bigsqcup\{([\![- \,;\, - \vdash \delta x.\, \delta t.\, \overline{C}_{\llcorner}\langle x_{\llcorner} t, t\rangle : (\neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\, [\,]\, [\,])^n \bot \mid n \geq 0\} \\
&= \bigsqcup\{([\![- \,;\, - \vdash \delta x.\, \delta t.\, \overline{C}_{\llcorner}\langle x_{\llcorner} t, t\rangle : (\neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\, [\,]\, [\,])^n \bot\, k \mid n \geq 0\} \\
&\quad \text{by continuity of } \lambda f \in [\![- \vdash \neg\mathbf{S} \multimap \neg\mathbf{S}]\!].\, f\, k
\end{aligned}$$

and

$$\begin{aligned}
&\bigsqcup\{([\![t : \neg\mathbf{S} \,;\, - \vdash \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S} \to \neg\mathbf{S}]\!]\, [\,]\, [[\,] \mid t{:}\, k])^n \bot \mid n \geq 0\} \\
&= [\![t : \neg\mathbf{S} \,;\, - \vdash \mathsf{Y}^\circ\, \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S} \to \neg\mathbf{S}]\!]\, [\,]\, [[\,] \mid t{:}\, k] \\
&\quad \text{by Lemma 35}
\end{aligned}$$

and so the result follows from

$$\begin{aligned}
&\bigsqcup\{([\![- \,;\, - \vdash \delta x.\, \delta t.\, \overline{C}_{\llcorner}\langle x_{\llcorner} t, t\rangle : (\neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\, [\,]\, [\,])^n \bot\, k \mid n \geq 0\} \\
&= \bigsqcup\{([\![t : \neg\mathbf{S} \,;\, - \vdash \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S} \to \neg\mathbf{S}]\!]\, [\,]\, [[\,] \mid t{:}\, k])^n \bot \mid n \geq 0\}
\end{aligned}$$

which is implied by

$$\begin{aligned}
&\{([\![- \,;\, - \vdash \delta x.\, \delta t.\, \overline{C}_{\llcorner}\langle x_{\llcorner} t, t\rangle : (\neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\, [\,]\, [\,])^n \bot\, k \mid n \geq 0\} \\
&= \{([\![t : \neg\mathbf{S} \,;\, - \vdash \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S} \to \neg\mathbf{S}]\!]\, [\,]\, [[\,] \mid t{:}\, k])^n \bot \mid n \geq 0\}
\end{aligned}$$

which, for any $n \geq 0$, is in turn implied by

$$\begin{aligned}
&([\![- \,;\, - \vdash \delta x.\, \delta t.\, \overline{C}_{\llcorner}\langle x_{\llcorner} t, t\rangle : (\neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\, [\,]\, [\,])^n \bot\, k \\
&= ([\![t : \neg\mathbf{S} \,;\, - \vdash \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S} \to \neg\mathbf{S}]\!]\, [\,]\, [[\,] \mid t{:}\, k])^n \bot
\end{aligned} \tag{8.9}$$

which we prove by induction on n:

$[n = 0]$: Immediate.

$[n = m + 1]$: Assume (8.9) for m. Then let

$$\begin{aligned}
e &= [[\,] \mid x{:}\, ([\![- \,;\, - \vdash \delta x.\, \delta t.\, \overline{C}_{\llcorner}\langle x_{\llcorner} t, t\rangle : (\neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\, [\,]\, [\,])^m \bot] \\
e' &= [e \mid t{:}\, k] \\
e'' &= [[[\,] \mid t{:}\, k] \mid l{:}\, ([\![t : \neg\mathbf{S} \,;\, - \vdash \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S} \to \neg\mathbf{S}]\!]\, [\,]\, [[\,] \mid t{:}\, k])^m \bot]
\end{aligned}$$

and we have

$$
\begin{aligned}
&(\llbracket - \,;\, - \vdash \delta x.\, \delta t.\, \overline{C}\llcorner\langle x \llcorner t, t\rangle : (\neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, [\,])^{m+1}\, \bot\, k \\
&= \llbracket - \,;\, - \vdash \delta x.\, \delta t.\, \overline{C}\llcorner\langle x \llcorner t, t\rangle : (\neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, [\,] \\
&\quad ((\llbracket - \,;\, - \vdash \delta x.\, \delta t.\, \overline{C}\llcorner\langle x \llcorner t, t\rangle : (\neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, [\,])^{m}\, \bot)\, k \\
&= \llbracket x : \neg\mathbf{S} \multimap \neg\mathbf{S} \,;\, t : \neg\mathbf{S} \vdash \overline{C}\llcorner\langle x \llcorner t, t\rangle : \neg\mathbf{S}\rrbracket\, [\,]\, e' \\
&= \llbracket x : \neg\mathbf{S} \multimap \neg\mathbf{S} \,;\, t : \neg\mathbf{S} \vdash \overline{C} : \neg\mathbf{S} \,\&\, \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, e' \\
&\quad (\llbracket x : \neg\mathbf{S} \multimap \neg\mathbf{S} \,;\, t : \neg\mathbf{S} \vdash \langle x \llcorner t, t\rangle : \neg\mathbf{S} \,\&\, \neg\mathbf{S}\rrbracket\, [\,]\, e') \\
&= \llbracket - \,;\, - \vdash \overline{C} : \neg\mathbf{S} \,\&\, \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, [\,]\, (\llbracket x : \neg\mathbf{S} \multimap \neg\mathbf{S} \,;\, t : \neg\mathbf{S} \vdash \langle x \llcorner t, t\rangle : \neg\mathbf{S} \,\&\, \neg\mathbf{S}\rrbracket\, [\,]\, e') \\
&\quad \text{by Proposition 22 and Lemma 33} \\
&= \llbracket - \,;\, - \vdash \overline{C} : \neg\mathbf{S} \,\&\, \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, [\,] \\
&\quad (\llbracket x : \neg\mathbf{S} \multimap \neg\mathbf{S} \,;\, t : \neg\mathbf{S} \vdash x \llcorner t : \neg\mathbf{S}\rrbracket\, [\,]\, e', \llbracket x : \neg\mathbf{S} \multimap \neg\mathbf{S} \,;\, t : \neg\mathbf{S} \vdash t : \neg\mathbf{S}\rrbracket\, [\,]\, e') \\
&= \llbracket - \,;\, - \vdash \overline{C} : \neg\mathbf{S} \,\&\, \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, [\,] \\
&\quad (\llbracket x : \neg\mathbf{S} \multimap \neg\mathbf{S} \,;\, - \vdash x : \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, e\, (\llbracket x : \neg\mathbf{S} \multimap \neg\mathbf{S} \,;\, t : \neg\mathbf{S} \vdash t : \neg\mathbf{S}\rrbracket\, [\,]\, e'), k) \\
&= \llbracket - \,;\, - \vdash \overline{C} : \neg\mathbf{S} \,\&\, \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, [\,] \\
&\quad ((\llbracket - \,;\, - \vdash \delta x.\, \delta t.\, \overline{C}\llcorner\langle x \llcorner t, t\rangle : (\neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, [\,])^{m}\, \bot\, k, k) \\
&= \llbracket - \,;\, - \vdash \overline{C} : \neg\mathbf{S} \,\&\, \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, [\,] \\
&\quad ((\llbracket t : \neg\mathbf{S} \,;\, - \vdash \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S} \to \neg\mathbf{S}\rrbracket\, [\,]\, [[\,] \mid t{:}k])^{m}\, \bot, k) \\
&\quad \text{by the induction hypothesis} \\
&= \llbracket - \,;\, - \vdash \overline{C} : \neg\mathbf{S} \,\&\, \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, [\,] \\
&\quad (\llbracket t : \neg\mathbf{S}\,,\, l : \neg\mathbf{S} \,;\, - \vdash l : \neg\mathbf{S}\rrbracket\, [\,]\, e'', \llbracket t : \neg\mathbf{S}\,,\, l : \neg\mathbf{S} \,;\, - \vdash t : \neg\mathbf{S}\rrbracket\, [\,]\, e'') \\
&= \llbracket - \,;\, - \vdash \overline{C} : \neg\mathbf{S} \,\&\, \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, [\,]\, (\llbracket t : \neg\mathbf{S}\,,\, l : \neg\mathbf{S} \,;\, - \vdash \langle l, t\rangle : \neg\mathbf{S} \,\&\, \neg\mathbf{S}\rrbracket\, [\,]\, e'') \\
&= \llbracket t : \neg\mathbf{S}\,,\, l : \neg\mathbf{S} \,;\, - \vdash \overline{C} : \neg\mathbf{S} \,\&\, \neg\mathbf{S} \multimap \neg\mathbf{S}\rrbracket\, [\,]\, e'' \\
&\quad (\llbracket t : \neg\mathbf{S}\,,\, l : \neg\mathbf{S} \,;\, - \vdash \langle l, t\rangle : \neg\mathbf{S} \,\&\, \neg\mathbf{S}\rrbracket\, [\,]\, e'') \\
&\quad \text{by Proposition 22 and Lemma 33} \\
&= \llbracket t : \neg\mathbf{S}\,,\, l : \neg\mathbf{S} \,;\, - \vdash \overline{C}\llcorner\langle l, t\rangle : \neg\mathbf{S}\rrbracket\, [\,]\, e'' \\
&= \llbracket t : \neg\mathbf{S}\,,\, l : \neg\mathbf{S} \,;\, - \vdash \overline{\overline{C}}\, l\, t : \neg\mathbf{S}\rrbracket\, [\,]\, e'' \\
&\quad \text{by Proposition 23 and Lemma 34} \\
&= \llbracket t : \neg\mathbf{S} \,;\, - \vdash \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S} \to \neg\mathbf{S}\rrbracket\, [\,]\, [[\,] \mid t{:}k] \\
&\quad ((\llbracket t : \neg\mathbf{S} \,;\, - \vdash \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S} \to \neg\mathbf{S}\rrbracket\, [\,]\, [[\,] \mid t{:}k])^{m}\, \bot) \\
&= (\llbracket t : \neg\mathbf{S} \,;\, - \vdash \lambda l.\, \overline{\overline{C}}\, l\, t : \neg\mathbf{S} \to \neg\mathbf{S}\rrbracket\, [\,]\, [[\,] \mid t{:}k])^{m+1}\, \bot \qquad \square
\end{aligned}
$$

8.5 Hacked Linear CPS Transformation

If linearity is insisted on, we can hack around the need for Weakening in (8.8). We do not put any conceptual meaning to this interpretation since, intuitively, discarding is part of the desired semantics; so this interpretation is simply a way to fool the type system. Additionally, this interpretation is based on linear use of something larger than control context. So, basically, we linearly use more than the control context in order to slip through the typing, very similarly to how we linearly used less than the control context to allow linear typing of `call/cc` in Section 5.4. The point, then, of presenting this linear interpretation is to demonstrate that too large a control context is also dangerous, and just to provide another instance where we can subvert the formalism if we do not pay careful attention to the conceptual issues.

Discarding linearly used control contexts is possible since the only time Weakening is needed is to discard the recursive delimited continuation when passing the toplevel continuation to the command. Since the delimited continuations we use all have no free linear identifiers, and since we can form an "empty" closed delimited continuation, we can keep each delimited continuation in a &-pair with the identity function at all times and just "use" the identity function in order to discard the delimited continuation.

So we have a type of "discardable delimited continuations"

$$\underbrace{((\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R})}_{\text{delimited continuation}} \,\&\, \underbrace{((\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R})}_{\text{identity function}}$$

which can still be used linearly. Using this, we can give a version of the body of (8.8) having type

$$(\neg\mathbf{S} \multimap \neg\mathbf{S}) \,\&\, (\neg\mathbf{S} \multimap \neg\mathbf{S}) \multimap \neg\mathbf{S} \multimap \neg\mathbf{S}$$

in which we use a discardable delimited continuation in place of a delimited continuation. Then the linear transformation is

$$\overline{\mathtt{1}\!:\!C} \stackrel{\text{def}}{=} \mathsf{Y}^{\&\mathsf{id}}\, \delta\langle x, i\rangle.\, \delta t.\, \overline{C}\llcorner\langle x \llcorner t, i \llcorner t\rangle \tag{8.10}$$

where

$$\mathsf{Y}^{\&\mathsf{id}} \stackrel{\text{def}}{=} \mathsf{Y}\, \lambda y.\, \lambda x.\, x\llcorner\langle y\, x, \delta q.\, q\rangle : (P \,\&\, (Q \multimap Q) \multimap P) \to P \tag{8.11}$$

$$\stackrel{\beta\eta}{=} \lambda x.\, x\llcorner\langle \mathsf{Y}^{\&\mathsf{id}}\, x, \delta q.\, q\rangle \tag{8.12}$$

is a sort of fixed-point combinator:

Lemma 25

$$\begin{aligned}&[\![- \,;\, - \vdash \mathsf{Y}^{\&\mathsf{id}} : (P \,\&\, (Q \multimap Q) \multimap P) \to P]\!]\,[\,]\,[\,]\\ =\, &[\![- \,;\, - \vdash \lambda x.\, \mathsf{Y}\, \lambda p.\, x\llcorner\langle p, \delta q.\, q\rangle : (P \,\&\, (Q \multimap Q) \multimap P) \to P]\!]\,[\,]\,[\,]\end{aligned}$$

Proof Straightforward calculation of the same flavor as in the proof of Proposition 24, but centers on an inductive proof of, for any $d \in [\![- \vdash T]\!]$, for all $n \geq 0$

$$\begin{aligned}&([\![- \,;\, - \vdash \lambda y.\, \lambda x.\, x\llcorner\langle y\, x, \delta q.\, q\rangle : ((T \to P) \to T \to P) \to T \to P]\!]\,[\,]\,[\,])^n\, \bot\, d\\ =\, &([\![x : T \,;\, - \vdash \lambda p.\, x\llcorner\langle p, \delta q.\, q\rangle : P \to P]\!]\,[\,]\,[[\,] \mid x\!:\! d])^n\, \bot\end{aligned}$$

where $T = P \,\&\, (Q \multimap Q) \multimap P$. □

Note that this lemma says that in the denotational semantics (which ignores linearity), $\mathsf{Y}^{\&\mathsf{id}}$ has the same meaning as a term which computes a recursive delimited continuation by explicitly discarding the delimited continuation we got upset with (8.8) for discarding. We did say this is a hack. In (8.11), the use of y is not restricted, and it is discarded. Conceptually this is not problematic though since y, having type

$$((\neg\mathbf{S} \multimap \neg\mathbf{S}) \,\&\, (\neg\mathbf{S} \multimap \neg\mathbf{S}) \multimap \neg\mathbf{S} \multimap \neg\mathbf{S}) \to \neg\mathbf{S} \multimap \neg\mathbf{S}$$

is essentially a fixed-point combinator for discardable delimited continuations.

The linear interpretation is clearly computationally adequate with respect to the affine one. (8.12) ensures that i in $\delta t.\, \overline{C}\llcorner\langle x \llcorner t, i \llcorner t\rangle$ of (8.10) is always bound to the linear identity function. Hence, $i \llcorner t \stackrel{\beta\eta}{=} t$, eliminating the only significant difference in the bodies of the recursion. Lemma 25 and Lemma 21 then ensure that $\mathsf{Y}^{\&\mathsf{id}}$ computes the same fixed-point in the first factor of the &-pair, eliminating the other point of concern. That is:

Proposition 26 (Adequacy) *The linear interpretation is computationally adequate with respect to the affine interpretation.*

Proof

$$\begin{aligned}
&[\![- \,;\, - \vdash \mathsf{Y}^{\&\mathsf{id}}\, \delta\langle x, i\rangle.\, \delta t.\, \overline{C}_{\sqcup}\langle x \llcorner t, i \llcorner t\rangle : \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\,[\,]\,[\,] && \text{in the linear language}\\
=\ &[\![- \,;\, - \vdash \mathsf{Y}\, \lambda x.\, \delta t.\, \overline{C}_{\sqcup}\langle x \llcorner t, t\rangle : \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\,[\,]\,[\,] && \text{by Lemma 25}\\
=\ &[\![- \,;\, - \vdash \mathsf{Y}^{\circ}\, \delta x.\, \delta t.\, \overline{C}_{\sqcup}\langle x \llcorner t, t\rangle : \neg\mathbf{S} \multimap \neg\mathbf{S}]\!]\,[\,]\,[\,] && \text{in the affine lang., by Lemma 21}
\end{aligned}$$

□

8.6 Conclusion

In this chapter we analyzed recursion explicit in the source language, as opposed to implicit in self-application of untyped source procedures. We interpreted commands using the type:

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{labeled}\\ \text{continuation}}} \;\&\; \underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{current}\\ \text{continuation}}} \;\multimap \mathbf{S} \to \mathbf{R}$$

which is unsurprising. The standard semantics of this source language involves recursive continuations, as opposed to the recursive continuation transformers in λ-calculus. But we saw that recursive continuations break linear typing, and so we used an (adequate) interpretation built from recursive continuation transformers. Unfortunately, the type of commands does not force our hand into using a type which restricts use of the control context when defining the recursive continuation transformers. So we had to take fixed-points at type:

$$((\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}) \multimap (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}$$

which required a fixed-point combinator with type $(P \multimap P) \to P$.

This interpretation required an affine, rather than linear, type system. However, we presented a conceptually suspect method of hacking the semantics into a linear system.

Chapter 9

Coroutines

In this chapter we turn our attention to coroutines, a control construct allowing significantly different control behavior than we have thus far considered. The crucial aspect of coroutines which makes them so different is that running a coroutine modifies its control state, which is reinstated the next time the coroutine is run. Semantically, this moves us away from considering control contexts consisting only of control environment, to considering control contexts consisting of control store. The primary result of this is that the continuations of the control store are upward. Typechecking this in a restricted type system leads us to an interpretation which crucially relies on delimited continuations, which allow us to impose different usage constraints on the different parts of the semantic continuations.

9.1 Source Language

We now extend the simple command language of Section 6.1 with a facility for running two commands in parallel as coroutines.

9.1.1 Syntax

We extend the syntax of Section 6.1 with the productions

C	$::= \cdots$	*commands*
	$\mid$ `swap`	swap coroutines
P	$::=$	*programs*
	$\mid C \mid C$	coroutines in parallel

9.1.2 Direct semantics

The informal semantics of a program $C_0 \mid C_1$ is that execution starts with C_0 and continues until a `swap` command is executed. Whenever `swap` is executed, the currently running coroutine is blocked and execution proceeds with the other coroutine. For example, executing

$$0 = 42; \texttt{swap}; 1 = 13 \mid 2 = 7; \texttt{swap} \tag{9.1}$$

will first cause location 0 to be assigned value 42, then 2 gets 7, and finally 1 gets 13. Control returns to the operating system as soon as one command finishes, that is, the coroutines race.

More precisely, we give a simple resumption semantics of the source language [Plo76, HP79, Rey98b]. The type of resumptions is

$$Q \stackrel{\text{def}}{=} \mu Q.\, \mathbf{S} + (\mathbf{S} \times (\mathbf{S} \to Q))$$

and commands are interpreted with the type

$$\mathbf{S} \to Q$$

Executing a command on a state can either complete without executing `swap`—in which case the semantics is simply the modified state—or execute part of the command, changing the state to some intermediate state, and then execute `swap`—in which case the semantics is the intermediate state together with the unexecuted command. That is, the semantics of commands is

$$\begin{aligned}
[\![C]\!] &\in \mathbf{S} \to Q \\
[\![n = E]\!]\, s &\stackrel{\text{def}}{=} inl\, [s \mid n\colon (\!|E|\!)\, s] \\
[\![\texttt{skip}]\!]\, s &\stackrel{\text{def}}{=} inl\, s \\
[\![C_0\,;\, C_1]\!]\, s &\stackrel{\text{def}}{=} [\![C_1]\!]_;\, ([\![C_0]\!]\, s) \\
[\![\texttt{if(}E\texttt{) \{}C\texttt{\}}]\!]\, s &\stackrel{\text{def}}{=} \begin{cases} [\![C]\!]\, s & \text{if } (\!|E|\!)\, s \stackrel{\beta\eta}{=} n \neq 0 \\ inl\, s & \text{if } (\!|E|\!)\, s \stackrel{\beta\eta}{=} 0 \end{cases} \\
[\![\texttt{swap}]\!]\, s &\stackrel{\text{def}}{=} inr\, (s, inl)
\end{aligned}$$

where

$$\begin{aligned}
(\cdot)_; &\in (\mathbf{S} \to Q) \to Q \to Q \\
f_;\, (inl\, s) &\stackrel{\text{def}}{=} f\, s \\
f_;\, (inr\, (s, q)) &\stackrel{\text{def}}{=} inr\, (s, \lambda s.\, f_;\, (q\, s))
\end{aligned}$$

is the auxiliary definition of the meaning of sequencing common in resumption semantics.

The semantics of programs is

$$\begin{aligned}
[\![P]\!] &\in \mathbf{S} \to \mathbf{S} \\
[\![C_0 \mid C_1]\!]\, s &\stackrel{\text{def}}{=} [\![C_1]\!]_|\, ([\![C_0]\!]\, s)
\end{aligned}$$

where

$$\begin{aligned}
(\cdot)_| &\in (\mathbf{S} \to Q) \to Q \to \mathbf{S} \\
f_|\, (inl\, s) &\stackrel{\text{def}}{=} s \\
f_|\, (inr\, (s, g)) &\stackrel{\text{def}}{=} g_|\, (f\, s)
\end{aligned}$$

Here we need another auxiliary, $(\cdot)_|$, since parallel composition is semantically similar to sequencing, which we will see in the continuation semantics.

9.2 Refined CPS Transformation

We present a CPS interpretation which uses control contexts linearly in three stages. First we extend the transformation of Section 6.4 to handle the `swap` command, but this interpretation does not admit a satisfactory interpretation of programs. So next we refine the result type, yielding an interpretation in which control contexts are represented by delimited continuations. This interpretation admits a satisfactory interpretation of programs, however an affine, rather than linear, type system is required. Finally, we employ a hack to define a type of linearly used but discardable delimited continuations, which yields a linear interpretation.

9.2.1 Continuation interpretation

It has been known for some time that the combination of state and labels can be used to implement coroutines [Rey70, Wan99],[1] but to design a continuation semantics of coroutines we do not need the full power of the features used in these encodings; namely, first-class control and higher-order store. But we need to do more than simply have several continuations, one for each coroutine, and swap them. The extra ingredient that is needed is the ability to pass the saved state of one coroutine to another, so the other coroutine can then swap back; this is implemented using upward continuations and a recursive type.

Expressions can still be interpreted with the semantics given in Section 6.3.2. The execution of a command in the extended source language possibly changes not only the state, as before, but also the control state of the blocked coroutine; so the type of command continuations is

$$K \stackrel{\text{def}}{=} \mu K.\, \mathbf{S} \to K \multimap \mathbf{R}$$

Note that a command continuation takes another command continuation as an argument, hence command continuations are upward. Commands are interpreted with the type

$$K \multimap K$$

Intuitively, the meaning of a command depends on both the commands following it, and on the other coroutine. For example, $0 = 42$ in (9.1) depends on `swap`; $1 = 13$ and $2 = 7$; `swap`. The control context then consists of both of these, which are represented as command continuations, and used linearly. Unfolding the type of commands once we have

$$\underbrace{K}_{\substack{\text{running}\\ \text{continuation}}} \multimap \mathbf{S} \to \underbrace{K}_{\substack{\text{blocked}\\ \text{continuation}}} \multimap \mathbf{R}$$

So the interpretation of a command accepts a continuation which represents the rest of the (current) coroutine, accepts a (data) state, accepts a continuation which is the control state of the other (blocked) coroutine, and then runs. This treatment is a form of state-passing style of a stored continuation for the control state of the blocked coroutine.

Note that the typing of this interpretation is very different from those in preceding chapters since the two argument continuations are passed independently,[2] rather than in a $\&$-pair. Here, when a continuation is invoked, another continuation is passed to it. This means that the invoked and argument continuations cannot come from the same $\&$-pair, and hence both current and blocked continuations must be used and cannot share a common ancestor continuation.

In Section 6.4, the type of command continuations is $\mathbf{S} \to \mathbf{R}$, so K here is simply the old type of command continuations, but with $K \multimap \mathbf{R}$ as the result type. So since the only command that will manipulate the blocked coroutine is `swap`, the transformation clauses of the other commands are simply linear η-expansions of those in Section 6.4, see Figure 9.1. Hence, we merely extend the transformation with a clause for `swap`:

$$\overline{\texttt{swap}} \stackrel{\text{def}}{=} \delta r.\, \lambda s.\, \delta b.\, b\, s\, \_\, r$$

This clause simply invokes the blocked continuation with the running continuation as the new blocked continuation, indicating that the effect of executing `swap` is simply to interchange running and blocked coroutines.

[1] Wand also notes "Another syntactic restriction which might be desirable is one which would prevent a continuation from being restarted more than once.", which this refined transformation may be seen as providing.

[2] Or, we could arrange the types to pass them in a $\otimes$-pair.

Notice how we could extend the continuation semantics without modifying, merely η-expanding, the previous clauses. This contrasts sharply with the monad-like resumption semantics where the addition of `swap` forces sequencing to change meaning, and forces tagging throughout the semantics. Also, the resumption semantics involves tagging; a complicated sequencing, $(\cdot)_;$; and trampolining at toplevel, $(\cdot)_!$, while the continuation semantics involves only primitives one is likely to find in an assembly language.

We would like to interpret programs with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

using the transformation

$$\overline{C_0 \mid C_1} \stackrel{\text{def}}{=} \delta t.\, \lambda s.\, \overline{C_0}\,\_\,(\lambda s.\, \delta b.\, t\, s)\, s\, \_\, (\overline{C_1}\,\_\,\lambda s.\, \delta b.\, t\, s)$$

but this requires duplication of the toplevel continuation, t. Conceptually, both coroutines do need access to the toplevel continuation since either may finish and relinquish control to the operating system. But, only one coroutine may be running at any point in time, so it would be sufficient to only give a coroutine access to the toplevel continuation when it is actually running. This can be accomplished by representing the control state of a coroutine with a delimited continuation rather than a continuation.

9.2.2 Delimited continuation interpretation

Just as in Section 5.1, the delimited continuation interpretation is derived from the continuation interpretation simply by using a more specified result type. So instead of using a type of command continuations, we use a type of command delimited continuations

$$K \stackrel{\text{def}}{=} \mu K.\, \mathbf{S} \to K \multimap (\mathbf{S} \to K \multimap \mathbf{R}) \multimap \mathbf{R}$$

This type was obtained from the type used in the continuation interpretation, $\mu K.\, \mathbf{S} \to K \multimap \mathbf{R}$, but using $(\mathbf{S} \to K \multimap \mathbf{R}) \multimap \mathbf{R}$ as the result type.

Commands are still interpreted with the type

$$K \multimap K$$

but now unfolding once yields

$$\underbrace{K}_{\substack{\text{running}\\ \text{delimited}\\ \text{continuation}}} \multimap \mathbf{S} \to \underbrace{K}_{\substack{\text{blocked}\\ \text{delimited}\\ \text{continuation}}} \multimap \underbrace{(\mathbf{S} \to K \multimap \mathbf{R})}_{\substack{\text{shared}\\ \text{continuation}}} \multimap \mathbf{R}$$

As in Section 5.1, the transformation clauses for the delimited continuation interpretation are simply linear η-expansions of the clauses for the continuation interpretation. But now it is no longer necessary to duplicate the toplevel continuation when we try to give an interpretation of programs using the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

Figure 9.1 Refined CPS of Coroutines

Expressions

$$\mathbf{S} \to \mathbf{N}$$

$$\begin{aligned}
(\!|n|\!) &\stackrel{\text{def}}{=} \lambda s.\, n \\
(\!|{*n}|\!) &\stackrel{\text{def}}{=} \lambda s.\, s[n] \\
(\!|E_0 == E_1|\!) &\stackrel{\text{def}}{=} \lambda s.\, (\!|E_0|\!)\, s \stackrel{n}{=} (\!|E_1|\!)\, s
\end{aligned}$$

Commands

$$\underbrace{K}_{\text{running}} \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mu K.\, \mathbf{S} \to \underbrace{K}_{\text{blocked}} \multimap \underbrace{(\mathbf{S} \to K \multimap \mathbf{R})}_{\text{shared}} \multimap \mathbf{R}$$

$$\begin{aligned}
\overline{n = E} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, k\,[s \mid n\colon (\!|E|\!)\, s] \\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta k.\, k \\
\overline{C_0\,;\, C_1} &\stackrel{\text{def}}{=} \delta k.\, \overline{C_0} \lrcorner (\overline{C_1} \lrcorner k) \\
\overline{\texttt{if}(E)\,\{C\}} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, (\!|E|\!)\, s \to k\, s \,[]\, \overline{C} \lrcorner k\, s \\
\overline{\texttt{swap}} &\stackrel{\text{def}}{=} \delta r.\, \lambda s.\, \delta b.\, b\, s \lrcorner r
\end{aligned}$$

Programs

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{S} \to \mathbf{R}$$

$$\begin{aligned}
\overline{C_0 \mid C_1} &\stackrel{\text{def}}{=} \delta t.\, \lambda s.\, \overline{C_0} \lrcorner \mathsf{E}\, s \lrcorner (\overline{C_1} \lrcorner \mathsf{E}) \lrcorner \lambda s.\, \delta b.\, t\, s \\
\mathsf{E} &\stackrel{\text{def}}{=} \lambda s.\, \delta b.\, \delta k.\, k\, s \lrcorner b
\end{aligned}$$

---

by:

$$\overline{C_0 \mid C_1} \stackrel{\text{def}}{=} \delta t.\, \lambda s.\, \overline{C_0} \lrcorner \mathsf{E}\, s \lrcorner (\overline{C_1} \lrcorner \mathsf{E}) \lrcorner \lambda s.\, \delta b.\, t\, s$$

where, for legibility, we use the abbreviation

$$\mathsf{E} \stackrel{\text{def}}{=} \lambda s.\, \delta b.\, \delta k.\, k\, s \lrcorner b$$

for the empty delimited continuation.

This transformation still discards the blocked delimited continuation when a coroutine finishes, however. This is seen by trying to derive

$$-\,;\, t : \mathbf{S} \to \mathbf{R} \vdash \lambda s.\, \delta b.\, t\, s : \mathbf{S} \to K \multimap \mathbf{R}$$

which requires Weakening for the restricted zone, [RWEAK], as in Section 8.2. As before, we can hack up a linear version.

9.2.3 Hacked linear delimited continuation interpretation

If one insists on banning Weakening, we can hack up a linear version of the previous interpretation. Similar to the situation in Section 8.5, this works by getting the control context wrong, and is possible since the only time Weakening is needed is to discard the blocked delimited continuation when the running coroutine finishes. So we have a type of discardable command delimited

continuations

$$K \stackrel{\text{def}}{=} \mu K.\, (\mathbf{S} \to K \multimap (\mathbf{S} \to K \multimap \mathbf{R}) \multimap \mathbf{R}) \,\&\, (\neg\mathbf{S} \multimap \neg\mathbf{S})$$

Commands are interpreted with the type

$$K \multimap K$$

which unfolded once is

$$\underbrace{K}_{\substack{\text{running}\\ \text{delimited}\\ \text{continuation}}} \multimap (\mathbf{S} \to \underbrace{K}_{\substack{\text{blocked}\\ \text{delimited}\\ \text{continuation}}} \multimap \underbrace{(\mathbf{S} \to K \multimap \mathbf{R})}_{\substack{\text{shared}\\ \text{continuation}}} \multimap \mathbf{R}) \,\&\, \underbrace{(\neg\mathbf{S} \multimap \neg\mathbf{S})}_{\text{identity}}$$

The transformation of commands is not conceptually different, but it must be modified to carry the identity functions around appropriately:

$$\begin{aligned}
\overline{E_0 = E_1} &\stackrel{\text{def}}{=} \delta\langle r, i\rangle.\, \langle \lambda s.\, r\, [s \mid (\!|E_0|\!)\, s \colon (\!|E_1|\!)\, s],\, i\rangle \\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta k.\, k \\
\overline{C_0\,;\, C_1} &\stackrel{\text{def}}{=} \delta k.\, \overline{C_0} \lrcorner (\overline{C_1} \lrcorner k) \\
\overline{\texttt{if}(E)\,\{C\}} &\stackrel{\text{def}}{=} \delta\langle r, i\rangle.\, \langle \lambda s.\, (\!|E|\!)\, s \to r\, s \,[\!]\, \pi_0 \lrcorner (\overline{C} \lrcorner \langle r, i\rangle)\, s,\, i\rangle \\
\overline{\texttt{swap}} &\stackrel{\text{def}}{=} \delta\langle r, i\rangle.\, \langle \lambda s.\, \delta\langle b, j\rangle.\, b\, s \lrcorner \langle r, i\rangle,\, i\rangle
\end{aligned}$$

Programs are interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

and the transformation is

$$\overline{C_0 \mid C_1} \stackrel{\text{def}}{=} \delta t.\, \lambda s.\, \pi_0 \lrcorner (\overline{C_0} \lrcorner \mathsf{E})\, s \lrcorner (\overline{C_1} \lrcorner \mathsf{E}) \lrcorner \lambda s.\, \delta\langle b, j\rangle.\, j \lrcorner t\, s$$

where, for legibility, we use the abbreviation

$$\mathsf{E} \stackrel{\text{def}}{=} \langle \lambda s.\, \delta k.\, \delta t.\, t\, s \lrcorner k,\, \delta t.\, t\rangle$$

for the empty delimited continuation.

9.3 Soundness

The soundness of the linear delimited continuation interpretation is much like previous cases.

Proposition 27 (Soundness) *1. For any command C*

$$- \,;\, - \vdash \overline{C} : K \multimap K$$

2. For any program P

$$- \,;\, - \vdash \overline{P} : (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{S} \to \mathbf{R}$$

Proof

1. By structural induction on the syntax of C.
2. By structural induction on the syntax of P. □

9.4 Adequacy

No particular difficulties arise when relating the direct and linear delimited continuation semantics.

Proposition 28 (Adequacy) *For any program P and state s, if $[\![P]\!]\, s = s'$, then for any target term M, $\overline{P}\,M\, s \overset{\beta\eta}{=} M\, s'$.*

For induction purposes, we define a notion of syntactic size for the source language:

$$\begin{aligned} |C_0 \mathtt{\ ;\ } C_1| &\overset{\text{def}}{=} |C_0| + |C_1| \\ |E_0 \mathtt{\ =\ } E_1| &\overset{\text{def}}{=} 1 \\ |\mathtt{skip}| &\overset{\text{def}}{=} 0 \\ |C_0 \mathtt{\ ;\ } C_1| &\overset{\text{def}}{=} 1 + |C_0| + |C_1| \\ |\mathtt{if(}E\mathtt{)\ \{}C\mathtt{\}}| &\overset{\text{def}}{=} 1 + |C| \\ |\mathtt{swap}| &\overset{\text{def}}{=} 1 \end{aligned}$$

We need the following observation about the direct semantics:

Lemma 29 *If $[\![C]\!]\, s = inr\,(s', q)$, then $q = [\![C']\!]$ for some C' such that $|C'| < |C|$.*

Proof By structural induction on the syntax of C. □

We also need the following observation about the CPS transformation:

Lemma 30 *For any C and M, $\pi_1\,(\overline{C}\,\langle M, \delta t.\, t\rangle) \overset{\beta\eta}{=} \delta t.\, t$.*

Proof By structural induction on the syntax of C. □

The following two lemmas capture the correspondence between the direct and continuation interpretations of commands:

Lemma 31 *If $[\![C]\!]\, s = inl\, s'$, then $\pi_0\,(\overline{C}\,M)\, s \overset{\beta\eta}{=} \pi_0\, M\, s'$.*

Proof Assume $[\![C]\!]\, s = inl\, s'$ and proceed by structural induction on the syntax of C:

$[E_0 \mathtt{\ =\ } E_1]$: Therefore $s' = [s \mid (\!|E_0|\!)\, s \colon (\!|E_1|\!)\, s]$ and we have

$$\begin{aligned} & \pi_0\,(\overline{E_0 \mathtt{\ =\ } E_1}\,M)\, s \\ = {} & \pi_0\,((\delta\langle r, i\rangle.\, \langle \lambda s.\, r\, [s \mid (\!|E_0|\!)\, s \colon (\!|E_1|\!)\, s], i\rangle)\,M)\, s \\ \overset{\beta\eta}{=} {} & \pi_0\,\langle \lambda s.\, \pi_0\, M\, [s \mid (\!|E_0|\!)\, s \colon (\!|E_1|\!)\, s], \pi_1\, M\rangle\, s \\ \overset{\beta\eta}{=} {} & (\lambda s.\, \pi_0\, M\, [s \mid (\!|E_0|\!)\, s \colon (\!|E_1|\!)\, s])\, s \\ \overset{\beta\eta}{=} {} & \pi_0\, M\, [s \mid (\!|E_0|\!)\, s \colon (\!|E_1|\!)\, s] \\ = {} & \pi_0\, M\, s' \end{aligned}$$

$[\mathtt{skip}]$: Therefore $s' = s$ and we have

$$\begin{aligned} & \pi_0\,(\overline{\mathtt{skip}}\,M)\, s \\ = {} & \pi_0\,((\delta k.\, k)\,M)\, s \\ \overset{\beta\eta}{=} {} & \pi_0\, M\, s \\ = {} & \pi_0\, M\, s' \end{aligned}$$

$[C_0\,;C_1]$: Therefore $[\![C_1]\!]_;\,([\![C_0]\!]\,s) = inl\,s'$ and hence $[\![C_0]\!]\,s = inl\,s''$ and $[\![C_1]\!]\,s'' = inl\,s'$ for some s''. Therefore we have

$$\begin{aligned}
&\pi_0 \llcorner (\overline{C_0\,;C_1} \llcorner M)\,s \\
=\ &\pi_0 \llcorner ((\delta k.\,\overline{C_0} \llcorner (\overline{C_1} \llcorner k)) \llcorner M)\,s \\
\overset{\beta\eta}{=}\ &\pi_0 \llcorner (\overline{C_0} \llcorner (\overline{C_1} \llcorner M))\,s \\
\overset{\beta\eta}{=}\ &\pi_0 \llcorner (\overline{C_1} \llcorner M)\,s'' && \text{by the induction hypothesis for } C_0 \\
\overset{\beta\eta}{=}\ &\pi_0 \llcorner M\,s' && \text{by the induction hypothesis for } C_1
\end{aligned}$$

$[\texttt{if(}E\texttt{) \{}C_0\texttt{\}}]$: There are two cases:

$[(\!|E|\!)\,s \overset{\beta\eta}{=} n \neq 0]$: Therefore $[\![C_0]\!]\,s = inl\,s'$ and we have

$$\begin{aligned}
&\pi_0 \llcorner (\overline{\texttt{if(}E\texttt{) \{}C_0\texttt{\}}} \llcorner M)\,s \\
=\ &\pi_0 \llcorner ((\delta\langle r,i\rangle.\,\langle \lambda s.\,(\!|E|\!)\,s \to r\,s \,[\!]\, \pi_0 \llcorner (\overline{C_0} \llcorner \langle r,i\rangle)\,s, i\rangle) \llcorner M)\,s \\
\overset{\beta\eta}{=}\ &\pi_0 \llcorner \langle \lambda s.\,(\!|E|\!)\,s \to \pi_0 \llcorner M\,s \,[\!]\, \pi_0 \llcorner (\overline{C_0} \llcorner M)\,s, \pi_1 \llcorner M\rangle\,s \\
\overset{\beta\eta}{=}\ &(\lambda s.\,(\!|E|\!)\,s \to \pi_0 \llcorner M\,s \,[\!]\, \pi_0 \llcorner (\overline{C_0} \llcorner M)\,s)\,s \\
\overset{\beta\eta}{=}\ &(\!|E|\!)\,s \to \pi_0 \llcorner M\,s \,[\!]\, \pi_0 \llcorner (\overline{C_0} \llcorner M)\,s \\
\overset{\beta\eta}{=}\ &n \to \pi_0 \llcorner M\,s \,[\!]\, \pi_0 \llcorner (\overline{C_0} \llcorner M)\,s \\
\overset{\beta\eta}{=}\ &\pi_0 \llcorner (\overline{C_0} \llcorner M)\,s \\
\overset{\beta\eta}{=}\ &\pi_0 \llcorner M\,s' && \text{by the induction hypothesis}
\end{aligned}$$

$[(\!|E|\!)\,s \overset{\beta\eta}{=} 0]$: Therefore $s' = s$ and we have

$$\begin{aligned}
&\pi_0 \llcorner (\overline{\texttt{if(}E\texttt{) \{}C_0\texttt{\}}} \llcorner M)\,s \\
\overset{\beta\eta}{=}\ &(\!|E|\!)\,s \to \pi_0 \llcorner M\,s \,[\!]\, \pi_0 \llcorner (\overline{C_0} \llcorner M)\,s \\
\overset{\beta\eta}{=}\ &0 \to \pi_0 \llcorner M\,s \,[\!]\, \pi_0 \llcorner (\overline{C_0} \llcorner M)\,s \\
\overset{\beta\eta}{=}\ &\pi_0 \llcorner M\,s \\
=\ &\pi_0 \llcorner M\,s'
\end{aligned}$$

$[\texttt{swap}]$: Not possible. □

Lemma 32 *If* $[\![C]\!]\,s = inr\,(s',q)$*, then* $\pi_0 \llcorner (\overline{C} \llcorner M)\,s \llcorner N \overset{\beta\eta}{=} \pi_0 \llcorner N\,s' \llcorner (\overline{C'} \llcorner M)$ *where* $[\![C']\!] = q$ *and* $|C'| < |C|$.

Proof Assume $[\![C]\!]\,s = inr\,(s',q)$ and proceed by structural induction on the syntax of C:

$[E_0 \texttt{ = } E_1]$: Not possible.

$[\texttt{skip}]$: Not possible.

$[C_0\,;C_1]$: Therefore $[\![C_1]\!]_;\,([\![C_0]\!]\,s) = inr\,(s',q)$. Proceed by cases on $[\![C_0]\!]\,s$:

$[inl\, s'']$: Therefore $[\![C_1]\!]_; ([\![C_0]\!]\, s) = [\![C_1]\!]\, s'' = inr\, (s', q)$. Therefore, by Lemma 29, there exists C_1' such that $[\![C_1']\!] = q$ and $|C_1'| < |C_1|$. Thus we have

$$
\begin{aligned}
& \pi_0 \llcorner (\overline{C_0\,;\, C_1} \llcorner M)\, s \llcorner N \\
= {} & \pi_0 \llcorner ((\delta k.\, \overline{C_0} \llcorner (\overline{C_1} \llcorner k)) \llcorner M)\, s \llcorner N \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner (\overline{C_0} \llcorner (\overline{C_1} \llcorner M))\, s \llcorner N \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner (\overline{C_1} \llcorner M)\, s'' \llcorner N && \text{by Lemma 31} \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner N\, s' \llcorner (\overline{C_1'} \llcorner M) && \text{by the induction hypothesis for } C_1
\end{aligned}
$$

$[inr\, (s'', q')]$: Therefore, by Lemma 29, there exists C_0' such that $[\![C_0']\!] = q'$ and $|C_0'| < |C_0|$. So $[\![C_1]\!]_; ([\![C_0]\!]\, s) = inr\, (s'', \lambda s.\, [\![C_1]\!]_; ([\![C_0']\!]\, s)) = inr\, (s'', [\![C_0'\,;\, C_1]\!]) = inr\, (s', q)$ and hence $s'' = s'$, $[\![C_0'\,;\, C_1]\!] = q$, and $|C_0'\,;\, C_1| < |C_0\,;\, C_1|$. Thus we have

$$
\begin{aligned}
& \pi_0 \llcorner (\overline{C_0\,;\, C_1} \llcorner M)\, s \llcorner N \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner (\overline{C_0} \llcorner (\overline{C_1} \llcorner M))\, s \llcorner N \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner N\, s' \llcorner (\overline{C_0'} \llcorner (\overline{C_1} \llcorner M)) && \text{by the induction hypothesis for } C_0 \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner N\, s' \llcorner (\overline{C_0'\,;\, C_1} \llcorner M)
\end{aligned}
$$

$[\texttt{if(}E\texttt{)\ \{}C_0\texttt{\}}]$: Therefore $(\!|E|\!)\, s \overset{\beta\eta}{=} n \neq 0$ and $[\![C_0]\!]\, s = inr\, (s', q)$. Therefore the induction hypothesis ensures

$$\pi_0 \llcorner (\overline{C_0} \llcorner M)\, s \llcorner N \overset{\beta\eta}{=} \pi_0 \llcorner N\, s' \llcorner (\overline{C_0'} \llcorner M) \tag{9.2}$$

where $[\![C_0']\!] = q$ and $|C_0'| < |C_0|$. Thus we have

$$
\begin{aligned}
& \pi_0 \llcorner (\overline{\texttt{if(}E\texttt{)\ \{}C_0\texttt{\}}} \llcorner M)\, s \llcorner N \\
= {} & \pi_0 \llcorner ((\delta \langle r, i \rangle.\, \langle \lambda s.\, (\!|E|\!)\, s \to r\, s \,[]\, \pi_0 \llcorner (\overline{C_0} \llcorner \langle r, i \rangle)\, s, i \rangle) \llcorner M)\, s \llcorner N \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner \langle \lambda s.\, (\!|E|\!)\, s \to \pi_0 \llcorner M\, s \,[]\, \pi_0 \llcorner (\overline{C_0} \llcorner M)\, s, \pi_1 \llcorner M \rangle\, s \llcorner N \\
\overset{\beta\eta}{=} {} & ((\!|E|\!)\, s \to \pi_0 \llcorner M\, s \,[]\, \pi_0 \llcorner (\overline{C_0} \llcorner M)\, s) \llcorner N \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner (\overline{C_0} \llcorner M)\, s \llcorner N \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner N\, s' \llcorner (\overline{C_0'} \llcorner M) && \text{by (9.2)}
\end{aligned}
$$

$[\texttt{swap}]$: Therefore $s' = s$, $q = inl = [\![\texttt{skip}]\!]$. Note that $|\texttt{skip}| < |\texttt{swap}|$, and we have

$$
\begin{aligned}
& \pi_0 \llcorner (\overline{\texttt{swap}} \llcorner M)\, s \llcorner N \\
= {} & \pi_0 \llcorner ((\delta \langle r, i \rangle.\, \langle \lambda s.\, \delta \langle b, j \rangle.\, b\, s \llcorner \langle r, i \rangle, i \rangle) \llcorner M)\, s \llcorner N \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner \langle \lambda s.\, \delta \langle b, j \rangle.\, b\, s \llcorner M, \pi_1 \llcorner M \rangle\, s \llcorner N \\
\overset{\beta\eta}{=} {} & (\lambda s.\, \delta \langle b, j \rangle.\, b\, s \llcorner M)\, s \llcorner N \\
\overset{\beta\eta}{=} {} & \delta \langle b, j \rangle.\, b\, s \llcorner M) \llcorner N \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner N\, s \llcorner M \\
\overset{\beta\eta}{=} {} & \pi_0 \llcorner N\, s \llcorner ((\delta k.\, k) \llcorner M) \\
= {} & \pi_0 \llcorner N\, s' \llcorner (\overline{\texttt{skip}} \llcorner M)
\end{aligned}
$$

□

Proof [Proposition 28] By induction on $|P|$. Let $C_0 \mid C_1 = P$ and assume $[\![C_0 \mid C_1]\!]\, s = s'$. Therefore $[\![C_1]\!]_{\mid}\, ([\![C_0]\!]\, s) = s'$. Proceed by cases on $[\![C_0]\!]\, s$:

$[inl\, s'']$: Therefore $s'' = s'$ and we have

$$
\begin{array}{ll}
& \overline{C_0 \mid C_1} \llcorner M\, s \\
= & (\delta t.\, \lambda s.\, \pi_0 \llcorner (\overline{C_0} \llcorner \mathsf{E})\, s \llcorner (\overline{C_1} \llcorner \mathsf{E}) \llcorner \lambda s.\, \delta\langle b, j\rangle.\, j \llcorner t\, s) \llcorner M\, s \\
\overset{\beta\eta}{=} & \pi_0 \llcorner (\overline{C_0} \llcorner \mathsf{E})\, s \llcorner (\overline{C_1} \llcorner \mathsf{E}) \llcorner \lambda s.\, \delta\langle b, j\rangle.\, j \llcorner M\, s \\
\overset{\beta\eta}{=} & \pi_0 \llcorner \mathsf{E}\, s' \llcorner (\overline{C_1} \llcorner \mathsf{E}) \llcorner \lambda s.\, \delta\langle b, j\rangle.\, j \llcorner M\, s & \text{by Lemma 31} \\
\overset{\beta\eta}{=} & (\lambda s.\, \delta k.\, \delta t.\, t\, s \llcorner k)\, s' \llcorner (\overline{C_1} \llcorner \mathsf{E}) \llcorner \lambda s.\, \delta\langle b, j\rangle.\, j \llcorner M\, s \\
\overset{\beta\eta}{=} & (\lambda s.\, \delta\langle b, j\rangle.\, j \llcorner M\, s)\, s' \llcorner (\overline{C_1} \llcorner \mathsf{E}) \\
\overset{\beta\eta}{=} & \pi_1 \llcorner (\overline{C_1} \llcorner \mathsf{E}) \llcorner M\, s' \\
\overset{\beta\eta}{=} & (\delta t.\, t) \llcorner M\, s' & \text{by Lemma 30} \\
\overset{\beta\eta}{=} & M\, s'
\end{array}
$$

$[inr\, (s'', q)]$: Therefore Lemma 32 ensures

$$\pi_0 \llcorner (\overline{C_0} \llcorner M)\, s \llcorner N \overset{\beta\eta}{=} \pi_0 \llcorner N\, s'' \llcorner (\overline{C_0'} \llcorner M) \tag{9.3}$$

where $[\![C_0']\!] = q$ and $|C_0'| < |C_0|$. So we have $[\![C_0']\!]_{\mid}\, ([\![C_1]\!]\, s'') = s'$ and proceed by cases on $[\![C_1]\!]\, s''$:

$[inl\, s''']$: Therefore $s''' = s'$ and we have

$$
\begin{array}{ll}
& \overline{C_0 \mid C_1} \llcorner M\, s \\
\overset{\beta\eta}{=} & \pi_0 \llcorner (\overline{C_0} \llcorner \mathsf{E})\, s \llcorner (\overline{C_1} \llcorner \mathsf{E}) \llcorner \lambda s.\, \delta\langle b, j\rangle.\, j \llcorner M\, s \\
\overset{\beta\eta}{=} & \pi_0 \llcorner (\overline{C_1} \llcorner \mathsf{E})\, s'' \llcorner (\overline{C_0'} \llcorner \mathsf{E}) \llcorner \lambda s.\, \delta\langle b, j\rangle.\, j \llcorner M\, s & \text{by (9.3)} \\
\overset{\beta\eta}{=} & \pi_0 \llcorner \mathsf{E}\, s' \llcorner (\overline{C_0'} \llcorner \mathsf{E}) \llcorner \lambda s.\, \delta\langle b, j\rangle.\, j \llcorner M\, s & \text{by Lemma 31} \\
\overset{\beta\eta}{=} & (\lambda s.\, \delta\langle b, j\rangle.\, j \llcorner M\, s)\, s' \llcorner (\overline{C_0'} \llcorner \mathsf{E}) \\
\overset{\beta\eta}{=} & \pi_1 \llcorner (\overline{C_0'} \llcorner \mathsf{E}) \llcorner M\, s' \\
\overset{\beta\eta}{=} & (\delta t.\, t) \llcorner M\, s' & \text{by Lemma 30} \\
\overset{\beta\eta}{=} & M\, s'
\end{array}
$$

$[inr\, (s''', q')]$: Therefore Lemma 32 ensures

$$\pi_0 \llcorner (\overline{C_1} \llcorner M)\, s'' \llcorner N \overset{\beta\eta}{=} \pi_0 \llcorner N\, s''' \llcorner (\overline{C_1'} \llcorner M) \tag{9.4}$$

where $[\![C_1']\!] = q'$ and $|C_1'| < |C_1|$. So we have $[\![C_1']\!]_{\mid}\, ([\![C_0']\!]\, s''') = s'$ and hence $[\![C_0' \mid C_1']\!]\, s''' = s'$. Also, $|C_0' \mid C_1'| < |C_0 \mid C_1|$, and so the induction hypothesis ensures

$$\overline{C_0' \mid C_1'} \llcorner M\, s''' \overset{\beta\eta}{=} M\, s' \tag{9.5}$$

and we have

$$
\begin{aligned}
& \overline{C_0 \mid C_1} \llcorner M\, s \\
\overset{\beta\eta}{=} & \ \pi_0 \llcorner (\overline{C_0} \llcorner \mathsf{E}) \ s \llcorner (\overline{C_1} \llcorner \mathsf{E}) \llcorner \lambda s.\, \delta\langle b, j\rangle .\ j \llcorner M\, s \\
\overset{\beta\eta}{=} & \ \pi_0 \llcorner (\overline{C_1} \llcorner \mathsf{E}) \ s'' \llcorner (\overline{C_0'} \llcorner \mathsf{E}) \llcorner \lambda s.\, \delta\langle b, j\rangle .\ j \llcorner M\, s && \text{by (9.3)} \\
\overset{\beta\eta}{=} & \ \pi_0 \llcorner (\overline{C_0'} \llcorner \mathsf{E}) \ s''' \llcorner (\overline{C_1'} \llcorner \mathsf{E}) \llcorner \lambda s.\, \delta\langle b, j\rangle .\ j \llcorner M\, s && \text{by (9.4)} \\
\overset{\beta\eta}{=} & \ \overline{C_0' \mid C_1'} \llcorner M\, s''' \\
\overset{\beta\eta}{=} & \ M\, s' && \text{by (9.5)}
\end{aligned}
$$

□

9.5 Conclusion

In this chapter we encountered our first upward continuations in a semantics which restricts control contexts. The upward nature is evidenced in the interpretation:

$$\underbrace{K}_{\substack{\text{running}\\ \text{delimited}\\ \text{continuation}}} \multimap K$$

where

$$K \overset{\text{def}}{=} \mu K.\, \mathbf{S} \to \underbrace{K}_{\substack{\text{blocked}\\ \text{delimited}\\ \text{continuation}}} \multimap \underbrace{(\mathbf{S} \to K \multimap \mathbf{R})}_{\text{shared continuation}} \multimap \mathbf{R}$$

by K being recursive.

We saw that since the use of continuations by coroutines is so stylized, linear typing is unproblematic, even though the upward (delimited) continuations which interpret the coroutines play the role of a control store. However, to interpret toplevel programs reasonably, we found that the use of delimited continuations here is crucial. Since there are two coroutines, each with its control state represented by a continuation, if undelimited continuations were used, then the toplevel continuation would have to be shared between them, which linear typing is not happy with. Instead we used delimited continuations which allowed us to separate out the toplevel continuation and pass it back and forth between the coroutines as they are active, avoiding the need to actually share it.

This semantics is affine since the control state of one coroutine must be discarded when the other terminates (though, as before, there is a hack into a linear system).

Chapter 10

Stored Labels and Commands

Given the source language control constructs we have considered thus far—a couple sorts of labels and an implicit form of control store—it is natural to ask what the situation is for stored labels and commands. These language features are also of interest since storable pointers to immutable code are a prominent characteristic of the output of closure conversion in a compiler.

In this chapter we consider various forms of stored labels: some for which storing the labels themselves (as part of the data context) suffices, whose semantics manipulate a control environment; and some which necessitate storing the continuations the labels refer to (as part of the control context), whose semantics manipulate a control store. We also consider a form of stored code whose semantics is similar to the control environment semantics in that only data is stored, and similar to the control store semantics in that code is accessed through a store. Presenting these semantics will not only address the question we opened with, but will also let us explore the distinctions between control environment and control store, downward control and upward control, and stored code and stored control. Very briefly, control environment, downward control, and stored code all readily admit refined interpretations, while control store, upward control, and stored control only do so in very specialized or restricted forms.

10.1 Semantic Landscape

Ignoring linearity for the moment, the semantics in Chapter 7 and Chapter 8 follow the general form of interpreting commands with the type

$$U \to K \to K \quad \text{where} \quad K \stackrel{\text{def}}{=} S \to \mathbf{R} \tag{10.1}$$

while the semantics in Chapter 9 follows the form

$$K \to K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mu K.\, S \to U \to \mathbf{R} \tag{10.2}$$

where S is some type interpreting the data store and U is some type interpreting the *label assignment* (despite not having any explicit labels in Chapter 9). Drawing on the similarity between a label assignment and a traditional store, we say that a label assignment maps label *L-values* to label *R-values*. Note that while the label assignment is always a map conceptually, often a function type is not used in the implementation. For instance, in Chapter 7 label L-values are target language identifiers and label R-values are continuations, so label assignments have type

$$U \stackrel{\text{def}}{=} \mathbin{\&} K \tag{10.3}$$

If we had a type, I say, of target language identifiers, (10.3) would be equivalent to[1]

$$U = I \to K$$

Commonly, the label assignment and the current continuation together constitute the control context. The two forms of semantic interpretation treat the label assignment, and hence control context, very differently: in (10.1) it is treated as an environment (it is an input, but not an output, of commands), and is downward; while in (10.2) it is treated as a store (as an argument to continuations, it is both an input and an output of commands), and is upward. (In this latter case, since continuations accept a label assignment as an argument, and since the type of label assignments often itself involves the type of continuations (as in (10.3), for instance), use of a recursive type is required in (10.2), even for some non-recursive languages.)

Generally, semantics which genuinely fit form (10.1) admit a refined interpretation. However, as we will find, the pickings are much slimmer when dealing with semantics of form (10.2), and only very specialized or restricted cases lead to a refined interpretation. Part of the reason for this disparity may be that the diversity of store-based control constructs does not approach that of environment-based constructs. Linear use of control contexts is wholly dependent on (particular) stylized or idiomatic usage, but store constructs tend to be very general and allow any usage whatsoever. For stored code, the semantics looks to be of form (10.2), but in this case rather than a label assignment, U is another data store and hence its use need not be restricted in a refined interpretation.

The crucial aspect of each source language we consider, which determines which of the forms above its semantics fits, is whether it is sufficient to store only the L-values of labels, or if storing the R-values is necessary. In the first case, a semantics like (10.1) may be used, while in the second, one like (10.2) is required. We discuss this distinction further as we present the various source languages, and in Section 10.8 we summarize and discuss all the semantics.

10.2 Stored Global Labels

The first control construct we interpret in the control environment form of semantics is stored global labels. In this case there is some fixed collection of labels bound globally. The association between label L-values and R-values is fixed—executing code cannot rebind or redefine labels—so it is sufficient to treat the label assignment as a constant environment. While the label environment is immutable, label L-values are storable. So the map from locations to continuations ($\mathbf{N} \to K$), given by the composition of the data store mapping locations to values ($\mathbf{N} \to \mathbf{N}$) and label environment mapping label L-values to continuations ($\mathbf{N} \to K$), can be updated. In this way, the storability of labels connects the data store and control environment.

The ability to store global, immutable, labels does not necessitate a control store because, since the labels cannot be rebound (they are global) nor redefined (they are immutable), the same label environment is good throughout a program's execution. So there is no need to store the label environment, it is simply a global constant. It is only necessary to store label L-values, which point to some code but do not contain any information about control flow, and hence are just data.

10.2.1 Source language

We extend the syntax of Section 6.1 with the productions

[1] Details on this sort of equivalence will be discussed in Section 10.2.2.

$$
\begin{array}{lll}
E ::= \cdots & & \textit{expressions} \\
\quad \mid \mathtt{l}_i & & \text{label} \\
C ::= \cdots & & \textit{commands} \\
\quad \mid \mathtt{goto}\, E & & \text{jump} \\
P ::= \{C;\, \{\mathtt{l}_i : C\}_{i=1}^{n}\} & & \textit{programs} \qquad 0 \leq n
\end{array}
$$

where i is a number.

For the sake of convenience, we have specified the set of labels to be $\{\mathtt{l}_i \mid i \text{ a number}\}$, that is, isomorphic to $\mathbf{N}$. We could have more identifier-like (programmer-defined) labels if the semantics carried around an environment mapping labels to their L-values. But by taking the route we have, a label's L-value is trivially determined by the syntax of the label, without needing such an environment. The downside is that this makes the syntax of labels semantically significant, and hence labels are not renamable.

Since the labels in this language are global, we have programs consisting of a block which binds all the labels. Unlike in most other cases, we allow multiple labels to be bound in programs since we do not allow blocks to be nested. For simplicity's sake, we do not consider recursion in this chapter, but we will in Chapter 11. So in the program

$$\{C_0\,;\, \{\mathtt{l}_i : C_i\}_{i=1}^{n}\}$$

the scope of each label $\mathtt{l}_i$ is the commands

$$\{C_0, \ldots, C_{i-1}\}$$

Also, a stored label is defined only within its scope, so labels which escape their scope through the store are undefined. For example, the jump in the program

$$\{1 = \mathtt{l}_0\,;\, \mathtt{l}_0 : \mathtt{goto} * 1\}$$

is erroneous because location 1 contains label $\mathtt{l}_0$, whose scope does not contain $\mathtt{goto} * 1$.

Executing a program block sets the label environment and then execution proceeds with the first command in the block. When executing the body of a block, storing a label, $\mathtt{l}_i$, results in the label's L-value, i, being stored. Since the label environment never changes (destructively or otherwise), it is unnecessary to store the label R-values. With storable labels come computed jumps: we at least have to be able to pull a label out of the store and jump to it, as in

$$\{1 = \mathtt{l}_0\,;\, \mathtt{goto} * 1\,;\, 2 = 13\,;\, \mathtt{l}_0 : 2 = 42\}$$

which first binds $\mathtt{l}_0$ to the control point starting with $2 = 42$, and then stores $\mathtt{l}_0$'s L-value, 0, in location 1, jumps to the label R-value associated with the contents of location 1, causing 42 to be stored in location 2.

10.2.2 Refined CPS transformation

First we extend the semantics of expressions given in Section 6.3.2 with a clause for labels:

$$(\!|\mathtt{l}_i|\!) \stackrel{\text{def}}{=} \lambda s.\, i$$

Since labels are immutable in this language, the semantics of expressions only needs to specify a label's L-value.

The starting point for interpreting commands is the interpretation of the language of forward jumps in Section 7.4, where we used the type

$$\underbrace{\&_{\mathcal{L}}\, K}_{\substack{\text{label}\\ \text{environment}}} \quad \& \quad \underbrace{K}_{\substack{\text{current}\\ \text{continuation}}} \quad \multimap K \tag{10.4}$$

where

$$K \stackrel{\text{def}}{=} \mathbf{S} \to \mathbf{R}$$

Interpreting the label environment with a $\&$-tuple relies on substitution (and projection) to select the continuation associated with a given label. Now that the source language supports possibly computed jumps, we need computed projections. So instead of a $\&$-tuple, we interpret the label environment with a function mapping label L-values to continuations:

$$\mathbf{N} \to K$$

Using functions in this way is more general than using $\&$-tuples—we could have interpreted exceptions, forward jumps, and so on similarly—but some complication is added. The restrictions the typing makes are the same in either case. As when constructing a $\&$-tuple, where all the factors must share the same restricted zone, when constructing such a function all the possible outputs must share the same restricted zone. Using the sugar of Section 6.3.1, we have

$$\frac{\dfrac{\Gamma\,;\,\Delta \vdash U : \mathbf{N} \to P \quad \Gamma\,;\,- \vdash N_0 : \mathbf{N} \quad \Gamma\,;\,\Delta \vdash M_0 : P}{\Gamma\,;\,\Delta \vdash [U \mid N_0 \colon M_0] : \mathbf{N} \to P} \qquad \Gamma\,;\,- \vdash N_1 : \mathbf{N} \quad \Gamma\,;\,\Delta \vdash M_1 : P}{\Gamma\,;\,\Delta \vdash [[U \mid N_0 \colon M_0] \mid N_1 \colon M_1] : \mathbf{N} \to P}$$

from which we see that M_0, M_1 (and U) must share Δ. Deconstruction is more obvious: instead of using π_i to take us from a $\&$-tuple to a single factor, we simply use application to take us from a $\mathbf{N} \to P$ to a single P.

So, similar to (10.4), but with a different implementation of the environment, commands are interpreted with the type

$$\underbrace{(\mathbf{N} \to K)}_{\substack{\text{label}\\ \text{environment}}} \quad \& \quad \underbrace{K}_{\substack{\text{current}\\ \text{continuation}}} \quad \multimap K$$

and the transformation is given in Figure 6.1. Since label L-values are numbers and we are not relying on substitution, it is not necessary to parameterize the transformation on a sequence of labels, as was done in Section 7.4. Other than the clause for `goto`, the transformation is not conceptually different from that in Section 7.4. For `goto` E, since we now have computed jumps, first we must evaluate E to a label L-value, $(\!|E|\!)\, s$, then lookup the label R-value (continuation) associated with the label L-value by the environment, $u[(\!|E|\!)\, s]$, and finally invoke the continuation with the current state, $u[(\!|E|\!)\, s]\, s$.

Programs, which establish the binding of labels, are interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \quad \multimap \mathbf{S} \to \mathbf{R}$$

Figure 10.1 Refined CPS of Stored Global Labels

Expressions

$$\mathbf{S} \to \mathbf{N}$$

$$\begin{aligned}
(\!|n|\!) &\stackrel{\text{def}}{=} \lambda s.\, n \\
(\!|{*}n|\!) &\stackrel{\text{def}}{=} \lambda s.\, s[n] \\
(\!|E_0 \texttt{ == } E_1|\!) &\stackrel{\text{def}}{=} \lambda s.\, (\!|E_0|\!)\, s \stackrel{n}{=} (\!|E_1|\!)\, s \\
(\!|\mathtt{l}_i|\!) &\stackrel{\text{def}}{=} \lambda s.\, i
\end{aligned}$$

Commands

$$\underbrace{(\mathbf{N} \to K)}_{\text{labels}} \,\&\, \underbrace{K}_{\text{current}} \multimap K$$

$$\begin{aligned}
\overline{n \texttt{ = } E} &\stackrel{\text{def}}{=} \delta\langle u, k\rangle.\, \lambda s.\, k\,[s \mid n\colon (\!|E|\!)\, s] \\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta\langle u, k\rangle.\, k \\
\overline{C_0 \texttt{ ; } C_1} &\stackrel{\text{def}}{=} \delta\langle u, k\rangle.\, \overline{C_0}\llcorner\langle u, \overline{C_1}\llcorner\langle u, k\rangle\rangle \\
\overline{\texttt{if(}E\texttt{) \{}C\texttt{\}}} &\stackrel{\text{def}}{=} \delta\langle u, k\rangle.\, \lambda s.\, (\!|E|\!)\, s \to k\, s \,[]\, \overline{C}\llcorner\langle u, k\rangle\, s \\
\overline{\texttt{goto}\, E} &\stackrel{\text{def}}{=} \delta\langle u, k\rangle.\, \lambda s.\, u[(\!|E|\!)\, s]\, s
\end{aligned}$$

Programs

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{S} \to \mathbf{R}$$

$$\begin{aligned}
\llcorner\{C_0 \texttt{ ; } \{\mathtt{l}_i\colon C_i\}_{i=1}^{n}\}\lrcorner &\stackrel{\text{def}}{=} \delta t.\, \overline{C_0}\llcorner\langle \mathsf{U}_0, \cdots \overline{C_n}\llcorner\langle \mathsf{U}_n, t\rangle \cdots \rangle \\
\mathsf{U}_i &\stackrel{\text{def}}{=} [\cdots [\lambda n.\, (\mathsf{Y}\, \lambda x.\, x)\llcorner t \mid i+1\colon \overline{C_{i+1}}\llcorner\langle \mathsf{U}_{i+1}, \mathsf{U}_{i+1}[i+2]\rangle] \cdots \mid n\colon \overline{C_n}\llcorner\langle \mathsf{U}_n, t\rangle] : \mathbf{N} \to K
\end{aligned}$$

by the transformation in Figure 6.1. The transformation of a program is similar to that of sequencing above, but an appropriate label environment, U_i, is provided for each command, C_i. Each label environment contains definitions of only those labels which are in scope, the remainder are undefined, $(\mathsf{Y}\, \lambda x.\, x)\llcorner t$. Each label environment binds each label L-value, $i+1$, to a continuation for the labeled command, C_{i+1}, with the appropriate label environment, U_{i+1}, and the continuation denoting the next command, C_{i+2} (that is, $\mathsf{U}_{i+1}[i+2]$), for the current continuation. The final command, C_n, gets the toplevel continuation as its current continuation. Note that there is no recursion involved in the definition of the U_i, the notation is simply a shorthand for writing out all the environments needed for a known value of n.

10.3 Stored Nested Labels

It is natural to consider a source language with (statically-bound) stored nested labels, given that it is the result of simply adding the ability to store labels to the source language of Chapter 7, and that it provides a close first-order analogue of `call/cc`. We interpret this language with a semantics which manipulates a control store. This may be slightly confusing (if one naturally thinks of labels as being distinct from their denotations) since there are no explicit control store operations in the source language, but the combination of static binding and storability of labels

necessitates a control store.

The key difference from the language of the previous section is that storing the L-values of labels no longer suffices, the R-values must be stored. When labels are global, storing the L-value of a label suffices since the definitions of the labels never change, and hence a label can never be used in an incompatible label environment. With nested labels, however, storing only a label's L-value is insufficient since this stored L-value may persist after the label goes out of scope. That is, a label can outlive the environment it depends on. Hence, it becomes necessary to store a label's R-value: in this case, a continuation, which is implicitly a label, together with a label environment, that is, a continuation closure. Since continuations are part of a control context, this language provides stored control.

10.3.1 Source language

The source language is like that in Chapter 7 but with an additional command to store labels and one to jump to such a stored label. We extend the syntax of Section 7.1 with the productions

$$\begin{array}{lll} C ::= \cdots & & \textit{commands} \\ \quad \mid n = l & & \text{store label} \\ \quad \mid \texttt{goto} * n & & \text{indirect jump} \end{array}$$

While it would make sense to simply extend expressions to include labels, as in Section 10.2.1, doing so would complicate the presentation of the semantics for little gain.

To illustrate the intended semantics of this language, note that executing

$$\texttt{\{\{\{\{1 = l}_0\texttt{; goto l}_2\ \texttt{l}_0\texttt{:\}; 7 = 13; goto l}_4\ \texttt{l}_2\texttt{:\}; goto *1 l}_0\texttt{:\}; 7 = 42 l}_4\texttt{:\}$$

results in a state where location 7 contains 13. This happens since when $\texttt{l}_0$ is loaded from location 1, $\texttt{l}_0$ refers to the binding which was in effect when it was stored, that is, the control point starting with $7 = 13$. So storing a label results in a control point being stored; in other words, it is necessary to store label R-values.

10.3.2 CPS transformation

The conceptual starting point for interpreting this language is the semantics of Section 7.4:

$$\&_{\mathbb{Z}}\, K \,\&\, K \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mathbf{S} \to \mathbf{R}$$

The significant difference between the source language of Section 7.1 and the present one is that labels, in addition to numbers, are storable. So, while we can still interpret commands with the type

$$\underbrace{\&_{\mathbb{Z}}\, K}_{\substack{\text{control} \\ \text{environment}}} \quad \& \quad \underbrace{K}_{\substack{\text{current} \\ \text{continuation}}} \quad \multimap K$$

we must extend the type of the store to accommodate labels. Not yet considering usage constraints on the control context, instead of representing the store with $\mathbf{S}$, we need something along the lines of[2]

$$\mathbf{N} \to \mathbf{N} \,\&\, K$$

[2]This &-product should really be a union, but for our purposes here a &-product will do.

But in the type of continuations it is convenient to split the store into separate data and control components, and uncurry, yielding

$$\mu K.\quad \underbrace{\mathbf{S}}_{\text{data store}} \quad \to \quad \underbrace{(\mathbf{N} \to K)}_{\text{control store}} \quad \to \mathbf{R}$$

(Note that since the continuations serving as the R-values of labels are passed to other continuations as part of the store, the semantics must manipulate upward continuations, and the type of continuations must be recursive.) Now, the control store is part of the control context, and so should be subjected to usage constraints. As explained in Section 10.2.2, linear use of elements of type

$$\mathbf{N} \to K$$

is equivalent to linear use of elements of type

$$\mathbin{\&} K$$

but, for the situation here, neither express appropriate constraints. If we were to represent the control store with $\mathbin{\&} K$, say, then dereferencing one location would necessarily result in destruction of the entire state, making it impossible to dereference another location. Instead we use a multiplicative product (treated in Section 10.9), yielding a semantics of form

$$\mathbin{\&} K \mathbin{\&} K \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mu K.\, \mathbf{S} \to \bigotimes_z K \multimap \mathbf{R}$$

(Note that, technically, for this type of continuations to be well-defined, the $\otimes$-product must be finitary. So we need to specify that $\mathbf{N}$ represents some finite set of numbers, say 32-bit integers with modular arithmetic, and let z denote the maximum such number.)[3] While not quite of form (10.2) due to the presence of a control environment, this semantics certainly contains a control store. (And we could very well implement the control environment with the control store if we insisted on making (10.2) fit.)

Since

$$\begin{aligned} K &= \mu K.\, \mathbf{S} \to \bigotimes K \multimap \mathbf{R} \\ &= \mathbf{S} \to \bigotimes (\mu K.\, \mathbf{S} \to \bigotimes K \multimap \mathbf{R}) \multimap \mathbf{R} \\ &= \mathbf{S} \to \mathbf{R}' \end{aligned}$$

this semantics is simply that of Section 7.4 with a more specified result type. Hence, the transformation clauses for the old commands—which do not manipulate the control store—are simply η-expansions of those in Section 7.4, and we need only attempt to transform the new commands:

$$\begin{aligned} \overline{n = l}_{\vec{l}} &= \delta\langle \vec{l}, k\rangle.\, \lambda s.\, \delta(\vec{u}).\, k\, s\llcorner[(\vec{u}) \mid n{:}\, l] \\ \overline{\texttt{goto}\ {*}n}_{\vec{l}} &= \delta\langle \vec{l}, k\rangle.\, \lambda s.\, \delta(\vec{u}).\, \boxed{u_n}\, s\llcorner(\boxed{\vec{u}}) \end{aligned}$$

where $\vec{u} \stackrel{\text{def}}{=} u_0, \ldots, u_z$.

While not incredibly important, strictly speaking, the $n = l$ clause requires Weakening since the continuation u_n is forgotten. Real failure, however, occurs when jumping to a stored label: `goto` $*n$. Here the problem is that part of the control store, u_n, is copied (used twice), highlighted

[3] Another option would be to retain an infinite number type and use the infinitary tensor product of linear dependent type theory.

above by the boxes. The conceptual problem here is that once something is dumped into the store there is no constraint forcing any form of stylized usage. This is manifested as the ability to jump to the contents of *n* in the control store after having done so before. If we were to disallow such behavior, we could achieve linearity, but this is decidedly opposed to the intended semantics of the source language. But in Section 10.7 we consider imposing such constraints, in a language in which they are possibly not quite so counterintuitive.

There is another violation of linearity in the assignment clause since both *l* and *k* come from the argument &-tuple, meaning that it must be copied. Changing the interpretation to avoid this failure (by using a ⊗-tuple rather than a &-tuple for the control environment) would not immediately violate the intended semantics of the source language, but would instead force the clause for blocks to break the typing by copying the current continuation.

10.4 Dynamically-Assigned Labels

We now generalize the source language of Section 10.2 in a different direction, to allow nested blocks of dynamically-assigned (or fluidly-bound) labels. While such a language may appear completely contrived at first glance, the essential components are quite familiar: exceptions. In Section 10.4.3 we briefly discuss this relationship. This language more fully exercises the semantics, and also makes the point that storing and nesting of labels together are not *inherently* incompatible with linearity. Storability and nestability are not themselves the crucial factors, that is, in Section 10.3, linearity breaks due to (not restricted enough) stored control, not the combination of storable and nested labels. We support this with a restricted interpretation of a language with stored nested labels but which does not provide stored control.

The basic idea is to avoid needing label closures, which we saw in Section 10.3 lead to trying to put continuations in the store, breaking linearity. We achieve this by changing from statically binding labels to dynamically assigning them.[4] A dynamic assignment performs a temporary assignment. That is, an assignment is performed, then some code is executed, and then the assignment is reverted. Hence, using dynamic assignment can provide a mutable environment, which is more structured, stylized, and idiomatic than general assignment, but less structured than static binding since all environment lookups reference the current environment. That is, free identifiers in values are looked up in the current environment, not necessarily the environment which was current when the value was created. In other words, closures are not necessary. Dynamic assignment in general is explained more thoroughly in [FWH92].

10.4.1 Source language

We extend the syntax of Section 6.1 with the productions

$E ::= \cdots$		*expressions*
	$\mid \texttt{l}_i$	label
$C ::= \cdots$		*commands*
	$\mid \texttt{goto}\, E$	jump
	$\mid \{C\ \texttt{l}_i\texttt{:}\}$	block

where *i* is a number.

[4] Dynamically-bound labels would also be an option, and essentially equivalent, but dynamic assignment is a better fit with computed jumps, since the initial environment must bind all the labels anyhow. Also, it is more natural to think of assignment than binding when the "bound identifiers" cannot be renamed.

With dynamically-assigned labels, when entering a block $\{C\ \mathtt{l}_i\mathtt{:}\}$, first the control point immediately following the block is assigned to $\mathtt{l}_i$, then C is executed, and then $\mathtt{l}_i$ is reverted to its previous value. As an example, note that executing

$$\{\{\{\{1 = \mathtt{l}_0\mathtt{;}\ \mathtt{goto}\ \mathtt{l}_2\ \ \mathtt{l}_0\mathtt{:}\}\mathtt{;}\ 7 = 13\mathtt{;}\ \mathtt{goto}\ \mathtt{l}_4\ \ \mathtt{l}_2\mathtt{:}\}\mathtt{;}\ \mathtt{goto} * 1\ \ \mathtt{l}_0\mathtt{:}\}\mathtt{;}\ 7 = 42\ \ \mathtt{l}_4\mathtt{:}\}$$

results in a state where location 7 contains 42, not 13 as it did in Section 10.3.1 where labels were bound statically. Since the binding made by a block is reverted when the block is exited, storing a label only requires the label's L-value to be stored, not the R-value it refers to at the time it is stored.

10.4.2 Refined CPS transformation

Label expressions are interpreted as in Section 10.2.2 and so commands are interpreted with the type

$$\underbrace{(\mathbf{N} \to K)}_{\substack{\text{label}\\ \text{environment}}}\ \&\ \underbrace{K}_{\substack{\text{current}\\ \text{continuation}}}\ \multimap K$$

where

$$K \stackrel{\text{def}}{=} \mathbf{S} \to \mathbf{R}$$

One point to note about this type is that the label environment is handled in a downward fashion: a command accepts an environment but, although it is mutable, a modified environment is not returned upward (accepted by the current continuation). This indicates that any changes made to the environment by executing a command are reverted when the command completes.

The transformation is given in Figure 10.2. All but the clause for blocks are identical to those in Section 10.2.2. Note the crucial aspect of the interpretation of $\mathtt{goto}\,E$: the destination of any jump is looked up in the label environment, never is it part of the value of E. The clause for blocks indicates that to execute $\{C\ \mathtt{l}_i\mathtt{:}\}$, C should be executed in an environment where $\mathtt{l}_i$ is bound to the current continuation of the block.

Again as in Section 10.2.2, programs are interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}}\ \multimap \mathbf{S} \to \mathbf{R}$$

The transformation, shown in Figure 10.2, simply indicates that the initial label environment maps all labels to a divergent continuation.

10.4.3 Dynamically-assigned labels as exceptions

Nested blocks of dynamically-assigned labels provide the basic control machinery of an exceptions mechanism. Establishing a new binding of a label for the duration of a block's execution and then reverting to the previous binding, as the blocks in the present source language do, is much the same as installing a new handler for a sort of exception for the duration of the execution of some body of code, as is common to exception mechanisms. For instance, recall the example program from Section 10.4.1:

$$\{\{\{\{1 = \mathtt{l}_0\mathtt{;}\ \mathtt{goto}\ \mathtt{l}_2\ \ \mathtt{l}_0\mathtt{:}\}\mathtt{;}\ 7 = 13\mathtt{;}\ \mathtt{goto}\ \mathtt{l}_4\ \ \mathtt{l}_2\mathtt{:}\}\mathtt{;}\ \mathtt{goto} * 1\ \ \mathtt{l}_0\mathtt{:}\}\mathtt{;}\ 7 = 42\ \ \mathtt{l}_4\mathtt{:}\}$$

We can write a very similar version of this program using OCaml's exception mechanism, see Figure 10.3, which, when executed, also leaves 42 in location 7 (`loc_7`).

Figure 10.2 Refined CPS of Dynamically-Assigned Labels

Expressions

$$\mathbf{S} \to \mathbf{N}$$

$$\begin{aligned}
(\!|n|\!) &\stackrel{\text{def}}{=} \lambda s.\, n \\
(\!|{*n}|\!) &\stackrel{\text{def}}{=} \lambda s.\, s[n] \\
(\!|E_0 \texttt{ == } E_1|\!) &\stackrel{\text{def}}{=} \lambda s.\, (\!|E_0|\!)\, s \stackrel{n}{=} (\!|E_1|\!)\, s
\end{aligned}$$

Commands

$$\underbrace{(\mathbf{N} \to K)}_{\text{labels}} \,\&\, \underbrace{K}_{\text{current}} \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mathbf{S} \to \mathbf{R}$$

$$\begin{aligned}
\overline{n = E} &\stackrel{\text{def}}{=} \delta\langle u, k\rangle.\, \lambda s.\, k\,[s \mid n\colon (\!|E|\!)\, s] \\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta\langle u, k\rangle.\, k \\
\overline{C_0\,;\, C_1} &\stackrel{\text{def}}{=} \delta\langle u, k\rangle.\, \overline{C_0}\llcorner u, \overline{C_1}\llcorner u, k\rangle\rangle \\
\overline{\texttt{if(}E\texttt{) \{}C\texttt{\}}} &\stackrel{\text{def}}{=} \delta\langle u, k\rangle.\, \lambda s.\, (\!|E|\!)\, s \to k\, s \,[\!]\, \overline{C}\llcorner u, k\rangle\, s \\
\overline{\texttt{goto}\,E} &\stackrel{\text{def}}{=} \delta\langle u, k\rangle.\, \lambda s.\, u\,((\!|E|\!)\, s)\, s \\
\overline{\texttt{\{}C\ \texttt{l}_i\texttt{:\}}} &\stackrel{\text{def}}{=} \delta\langle u, k\rangle.\, \overline{C}\llcorner\langle [u \mid i\colon k], k\rangle
\end{aligned}$$

Programs

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{S} \to \mathbf{R}$$

$$\llcorner C \lrcorner \stackrel{\text{def}}{=} \delta t.\, \overline{C}\llcorner\langle \lambda n.\, (\Upsilon\, \lambda x.\, x)\llcorner t, t\rangle$$

Figure 10.3 Dynamically-Assigned Labels in OCaml

```
exception L0
exception L2
exception L4

let loc_1 = ref (Failure "undefined")
let loc_7 = ref 0

let _ =
  try (try (try (try
                   loc_1 := L0; raise L2
                 with L0 -> loc_7 := 13; raise L4)
            with L2 -> raise !loc_1)
       with L0 -> loc_7 := 42; raise L4)
  with L4 -> ()
```

10.5 Stored Commands

To investigate a semantics which manipulates a code store, we consider a language with stored commands. The essential point of this interpretation is that code which is stored, unlike stored control, need not form part of the control context; code is only data.

In this section we consider a language with a command to store a command, $n = (C)$, and to execute such a stored command, `exec` $*n$. As an illustration of the difference between stored code and stored control, note that the intended semantics of $n = (C_0) ; C_1$ is to store (the code of) command C_0 in location n of the code store and then execute C_1. After storing C_0, executing `exec` $*n ; C_2$ causes C_0 followed by C_2 to be executed. That is, if execution reaches the end of a stored command, then the code following the most recently executed `exec` command is executed. The code of a command (for example, C_0) is itself just some data: a sequence of opcodes. So in our example, when C_0 is stored, since nothing is stored which identifies which command to execute should execution of C_0 fall off the end, only code, not control, is stored.

On the other hand, suppose we were to change the semantics so that executing $n = (C_0) ; C_1$ followed by `exec` $*n ; C_2$ causes C_0 followed by C_1 to be executed. That is, if execution reaches the end of a stored command, then the code following the stored command in the program text is executed. In this case, not only C_0, but also C_1 (and whatever comes after it, ad infinitum), must be stored. All this together specifies a control point: executing `exec` $*n ; C_2$ completely ignores C_2 since the contents of n specifies a complete computation.

10.5.1 Source language

We extend the syntax of Section 6.1 with the productions

$$\begin{array}{lll} C ::= \cdots & & \textit{commands} \\ \quad \mid n = (C) & & \text{store command} \\ \quad \mid \texttt{exec} * n & & \text{execute stored command} \end{array}$$

For an example illustrating the intended semantics, the command:

$$1 = (\texttt{exec} * 2) ; 2 = (0 = *0 + 1) ; \texttt{exec} * 1$$

is simply a long-winded way to increment the value stored in location 0. Note that since the stored commands are inert until `exec` ed, and since a command *store* not a command *environment* is used, storing a command which refers to an uninitialized location is unproblematic.

10.5.2 Refined CPS transformation

The interpretation of commands is an extension of that in Section 6.4, so commands are interpreted with the type

$$\underbrace{K}_{\substack{\text{current} \\ \text{continuation}}} \multimap K$$

The key to the interpretation of stored commands is that in the type of continuations

$$K \stackrel{\text{def}}{=} \mu K.\ \underbrace{\mathbf{S}}_{\text{data store}} \to \underbrace{(\mathbf{N} \to K \multimap K)}_{\text{code store}} \to \mathbf{R}$$

the code store is not used linearly, since it is part of the data context. In Section 6.4, the type of continuations is $\mathbf{S} \to \mathbf{R}$, so K here is simply the old type of continuations, but with $(\mathbf{N} \to K \multimap K) \to \mathbf{R}$ as the result type. So we need only extend the transformation with clauses for the new commands:

$$\overline{n = (C)} \stackrel{\text{def}}{=} \delta k.\, \lambda s.\, \lambda t.\, k\, s\, [t \mid n{:}\overline{C}]$$
$$\overline{\texttt{exec}\ {*}n} \stackrel{\text{def}}{=} \delta k.\, \lambda s.\, \lambda t.\, t[n]_{\sqcup} k\, s\, t$$

Notice that $\overline{C}$ is stored without the current continuation, and that when executing a stored command, it is provided with the current continuation of the `exec` command.

Programs are interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

by the transformation, shown in Figure 10.4, simply indicates that the initial code store maps all locations to a divergent command, and that the code store is thrown away just before a program terminates.

Figure 10.4 Refined CPS of Stored Commands

Expressions

$$\mathbf{S} \to \mathbf{N}$$

$$\begin{aligned}
(\!|n|\!) &\stackrel{\text{def}}{=} \lambda s.\, n \\
(\!|{*n}|\!) &\stackrel{\text{def}}{=} \lambda s.\, s[n] \\
(\!|E_0 == E_1|\!) &\stackrel{\text{def}}{=} \lambda s.\, (\!|E_0|\!)\, s \stackrel{n}{=} (\!|E_1|\!)\, s
\end{aligned}$$

Commands

$$\underbrace{K}_{\text{current}} \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mu K.\ \underbrace{\mathbf{S}}_{\text{data store}} \to \underbrace{(\mathbf{N} \to K \multimap K)}_{\text{code store}} \to \mathbf{R}$$

$$\begin{aligned}
\overline{n = E} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, k\,[s \mid n\colon (\!|E|\!)\, s] \\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta k.\, k && \stackrel{\beta\eta}{=} \delta k.\, \lambda s.\, k\, s \\
\overline{C_0\,;\, C_1} &\stackrel{\text{def}}{=} \delta k.\, \overline{C_0}\llcorner(\overline{C_1}\llcorner k) && \stackrel{\beta\eta}{=} \delta k.\, \lambda s.\, \overline{C_0}\llcorner(\lambda s.\, \overline{C_1}\llcorner k\, s)\, s \\
\overline{\texttt{if}(E)\ \{C\}} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, (\!|E|\!)\, s \to k\, s \,[]\, \overline{C}\llcorner k\, s \\
\overline{n = (C)} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, \lambda t.\, k\, s\,[t \mid n\colon \overline{C}] \\
\overline{\texttt{exec}\ {*n}} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, \lambda t.\, t[n]\llcorner k\, s\, t
\end{aligned}$$

Programs

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{S} \to \mathbf{R}$$

$$\llcorner C \lrcorner \stackrel{\text{def}}{=} \delta k.\, \lambda s.\, \overline{C}\llcorner(\lambda s.\, \lambda t.\, k\, s)\, s\, \lambda n.\, \mathsf{Y}\, \lambda x.\, x$$

10.6 Mutable Labels, Natural but Nonlinear

In Section 10.3 we considered implicit control store, and in Section 10.5 we considered explicit code store, now we turn to explicit control store. This language does not really have a hope of a linear interpretation; we present it primarily to set up the next section, where we investigate what constraints on the source language are needed for a refined interpretation. But this language also demonstrates how even a very weak form of control store breaks linearity. When considering stored control, distinctions which were important to languages interpreted with control environments, such as storable versus unstorable and nested versus global, seem to be irrelevant: even the least powerful combination (unstorable global, but mutable, labels), which we present here, breaks linearity.

10.6.1 Source language

We extend the syntax of Section 6.1 with the productions

$C ::=$	$\cdots$	*commands*
	$\mid \texttt{l}_j \texttt{ = l}_i$	label assignment
	$\mid \texttt{goto l}_i$	jump
	$\mid \{C\ \texttt{l}_i\texttt{:}\}$	block

where $0 \leq i, j < n$ for some fixed n.

While the discussion has been in terms of label L-values from the beginning, we finally have labels on the left of an assignment. Intuitively, a label assignment $\texttt{l}_j = \texttt{l}_i$ works like ordinary assignment, $\texttt{l}_j$'s L-value is updated to $\texttt{l}_i$'s R-value, except that it refers to the control store rather than the data store. Note in particular that, unlike other languages, labels here are mutable but not storable. A consequence of this is that label L-values are not stored anywhere, only label R-values are stored (in the control store). $\texttt{goto l}_i$ also behaves as expected, looking up $\texttt{l}_i$'s R-value and invoking it. While it might look like a sort of binding, a block $\{C\ \texttt{l}_i\texttt{:}\}$ is just a form of forward-referencing assignment. That is, executing $\{C\ \texttt{l}_i\texttt{:}\}$ first assigns the control point corresponding to the end of the block to $\texttt{l}_i$ and then executes C.

To illustrate this language, we sketch an implementation of coroutines as in Chapter 9. We use $\texttt{l}_0$ to store the control state of the blocked coroutine, $\texttt{l}_1$ as an auxiliary, and $\texttt{l}_2$ to enable jumping to the end of the program and returning to the toplevel. A coroutine program $C_0 \mid C_1$ can be implemented by

$$\{\{C_0'\texttt{; goto l}_2\ \texttt{l}_0\texttt{:}\}\texttt{;}\ C_1'\texttt{; goto l}_2\ \texttt{l}_2\texttt{:}\}$$

Executing this first stores the control point corresponding to the toplevel continuation in $\texttt{l}_2$. Then the control point starting with $C_1'\texttt{; goto l}_2$ is stored in $\texttt{l}_0$. C_0' is then executed and, if it completes, $\texttt{l}_2$ is jumped to, causing the program to return to the toplevel. (Note how $\texttt{l}_2$ is passed along statefully, but unmodified, throughout the computation, just as the toplevel continuation in the delimited continuation interpretation of Section 9.2.2.) Above, C_0', C_1' are the versions of C_0, C_1 which have had occurrences of `swap` replaced by a command, described below, which swaps the coroutines by manipulating $\texttt{l}_0$ and the control point starting with the command following `swap`. So if executing C_0' results in a `swap` command being executed, the remainder of C_0' is stored in $\texttt{l}_0$ and the previous contents of $\texttt{l}_0$ are executed.

As mentioned above, occurrences of `swap` are replaced by

$$\texttt{l}_1 = \texttt{l}_0\texttt{; \{goto l}_1\ \texttt{l}_0\texttt{:\}}$$

Executing this command first moves the contents of $\texttt{l}_0$, the control state of the blocked coroutine, to $\texttt{l}_1$. Then the block is executed, which stores the control state of the "next" command, which might be thought of as the program counter plus one, in $\texttt{l}_0$. Finally, the previously blocked coroutine is resumed by jumping to $\texttt{l}_1$.

10.6.2 CPS transformation

The interpretation of commands is an extension of that in Section 6.4 but ignoring linearity, so commands are interpreted with the type

$$\underbrace{K}_{\substack{\text{current}\\ \text{continuation}}} \quad \to K$$

In Section 6.4 the type of continuations was $\mathbf{S} \to \mathbf{R}$, and here we simply further specify the result type to include a control store, yielding

$$K \stackrel{\text{def}}{=} \mu K.\ \underbrace{\mathbf{S}}_{\text{data store}} \quad \to \quad \underbrace{\&_n K}_{\text{control store}} \quad \to \mathbf{R}$$

where n is the number of labels, as specified in Section 10.6.1.

Instead of implementing the control store with $\&_\mathbb{Z} K$, we could have used $\mathbf{N} \to K$, but since the present language does not include computed jumps, it is not necessary. Also, we choose to use $\&_\mathbb{Z} K$ since it is closer to the interpretation we will give in the next section.

The transformation clauses for the new commands are

$$\begin{aligned}
\overline{\overline{\texttt{l}_j \texttt{ = l}_i}} &\stackrel{\text{def}}{=} \lambda k.\, \lambda s.\, \lambda\langle\vec{l}\rangle.\, k\, s\, [\langle\vec{l}\rangle \mid j\colon l_i] \\
\overline{\overline{\texttt{goto l}_i}} &\stackrel{\text{def}}{=} \lambda k.\, \lambda s.\, \lambda\langle\vec{l}\rangle.\, l_i\, s\, \langle\vec{l}\rangle \\
\overline{\overline{\texttt{\{}C\ \texttt{l}_i\texttt{:\}}}} &\stackrel{\text{def}}{=} \lambda k.\, \lambda s.\, \lambda\langle\vec{l}\rangle.\, \overline{\overline{C}}\, k\, s\, [\langle\vec{l}\rangle \mid i\colon k]
\end{aligned}$$

where $\vec{l} \stackrel{\text{def}}{=} l_0, \ldots, l_{n-1}$.

In the $\texttt{l}_j \texttt{ = l}_i$ clause, the current continuation is invoked with an unmodified data state and a control state like the old but where $\texttt{l}_j$'s L-value, j, maps to $\texttt{l}_i$'s R-value, l_i. For $\texttt{goto l}_i$, $\texttt{l}_i$'s R-value, l_i, is invoked with unchanged states. The clause for blocks, $\texttt{\{}C\ \texttt{l}_i\texttt{:\}}$, indicates that the body, C, should be executed with the same current continuation and data state, but in a control state like the old but where $\texttt{l}_i$'s L-value, i, maps to the current continuation, k.

Programs are interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel} \\ \text{continuation}}} \to \mathbf{S} \to \mathbf{R}$$

by the transformation

$$\llcorner C \lrcorner \stackrel{\text{def}}{=} \lambda t.\, \lambda s.\, \overline{\overline{C}}\, (\lambda s.\, \lambda\langle\vec{l}\rangle.\, t\, s)\, s\, \langle \{\mathsf{D}\}_0^{n-1} \rangle$$

where

$$\mathsf{D} \stackrel{\text{def}}{=} \mathsf{Y}\, \lambda k.\, k : K$$

simply indicating that the initial control store maps all labels to a divergent continuation, and the control store is thrown away just before termination.

10.7 Mutable Labels, Affine but Contrived

In this section we modify the source language of the previous section in order to force it to admit a refined interpretation. In doing do we admittedly end up with a contrived language, but it helps in understanding what a refined interpretation demands of the source language. The result of the restrictions allowing the refined interpretation is that all the operations on the control store must be very careful to move, not copy, stored pieces of control context.

10.7.1 Source language

While the syntax is the same as in the previous section, the intended semantics is significantly different. Now the effect of executing a label assignment $\texttt{l}_j \texttt{ = l}_i$ is not only to update $\texttt{l}_j$'s L-value with $\texttt{l}_i$'s R-value, as before, but also to undefine $\texttt{l}_i$'s L-value. Hence, $\texttt{l}_j$'s R-value is moved, not copied, from $\texttt{l}_i$'s L-value to $\texttt{l}_j$'s. Similarly, executing $\texttt{goto l}_i$ now not only looks up $\texttt{l}_i$'s R-value and invokes it, but also undefines $\texttt{l}_i$'s L-value. Along the same lines but stranger, executing a block $\texttt{\{}C\ \texttt{l}_i\texttt{:\}}$ first assigns the control point corresponding to the end of the block to $\texttt{l}_i$ and then executes C, but with an undefined current continuation. This means that the only sensible (defined) choices of C are of the form $C'\texttt{; goto l}_j$.

Note that while these restrictions are severe, this language is still expressive enough to implement coroutines as described in Section 10.6.1.

10.7.2 Refined CPS transformation

The only difference between the form of this interpretation and that in Section 10.6.2 is the use of a $\otimes$-product rather than a $\&$-product, to implement the control store. Hence, commands are interpreted with the type

$$\underbrace{K}_{\substack{\text{current}\\ \text{continuation}}} \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mu K.\ \underbrace{\mathbf{S}}_{\substack{\text{data}\\ \text{store}}} \to \underbrace{\bigotimes_n K}_{\substack{\text{control}\\ \text{store}}} \multimap \mathbf{R}$$

The transformation of commands, shown in Figure 10.5, is like that in Section 10.6.2 but for the introduction of undefinedness. In the $\mathtt{l}_j = \mathtt{l}_i$ clause, the current continuation is invoked with an unmodified data state and a control state like the old but where $\mathtt{l}_j$'s L-value, j, maps to $\mathtt{l}_i$'s R-value, l_i, and $\mathtt{l}_i$'s L-value is undefined (and consumes $\mathtt{l}_j$'s R-value, l_j). For $\mathtt{goto}\,\mathtt{l}_i$, $\mathtt{l}_i$'s R-value, l_i, is invoked with an unchanged data state and a control state like the old but where $\mathtt{l}_i$'s L-value is undefined. The clause for blocks, $\{C\ \mathtt{l}_i\mathtt{:}\}$, indicates that the body, C, should be executed with an undefined current continuation, and the same data state, but in a control state like the old but where $\mathtt{l}_i$'s L-value, i, maps to the current continuation, k. In short, all operations ensure that parts of the control state are moved and not copied.

Programs are interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

by the transformation in Figure 10.5, which indicates that the initial control store maps all labels to a divergent continuation, and the control store is thrown away just before termination. The more complicated definition of divergence is necessary to ensure continuations are used affinely when defining a recursive continuation directly.

10.7.3 Stored mutable labels

This semantics could be extended to allow the labels to be stored. This would give the source language code pointers, but as above, the code they point to could not be copied, only moved. One use of this would be to implement coroutines by interchanging the contents of two variables holding labels instead of swapping labels, as in Section 10.6.1. While this semantics is technically complicated, the same intuitive concepts and restrictions drive the definition. The complication is that, like we needed to interpret label environments with

$$\mathbf{N} \to K$$

instead of

$$\& K$$

with the addition of stored labels, we would need the linear dependent type theory analogue of

$$\bigotimes K$$

to interpret label stores once stored labels are added. For this reason we do not pursue this further.

Figure 10.5 Refined CPS of Mutable Labels

Expressions

$$\mathbf{S} \to \mathbf{N}$$

$$\begin{aligned}
(\!|n|\!) &\stackrel{\text{def}}{=} \lambda s.\, n \\
(\!|{*n}|\!) &\stackrel{\text{def}}{=} \lambda s.\, s[n] \\
(\!|E_0 \texttt{ == } E_1|\!) &\stackrel{\text{def}}{=} \lambda s.\, (\!|E_0|\!)\, s \stackrel{n}{=} (\!|E_1|\!)\, s
\end{aligned}$$

Commands

$$\underbrace{K}_{\text{current}} \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mu K.\ \underbrace{\mathbf{S}}_{\text{data store}} \to \underbrace{\textstyle\bigotimes_n K}_{\text{control store}} \multimap \mathbf{R}$$

$$\begin{aligned}
\overline{n = E} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, k\,[s \mid n\colon (\!|E|\!)\, s] \\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta k.\, k \\
\overline{C_0\,;\, C_1} &\stackrel{\text{def}}{=} \delta k.\, \overline{C_0}\llcorner(\overline{C_1}\llcorner k) \\
\overline{\texttt{if(}E\texttt{) \{}C\texttt{\}}} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, (\!|E|\!)\, s \to k\, s \,[]\, \overline{C}\llcorner k\, s \\
\overline{\texttt{l}_j\texttt{ = l}_i} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, \delta(\vec{l}).\, k\, s\llcorner[(\vec{l}) \mid i\colon \mathsf{D}\llcorner l_j] \mid j\colon l_i] \\
\overline{\texttt{goto l}_i} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, \delta(\vec{l}).\, l_i\, s\llcorner[(\vec{l}) \mid i\colon \mathsf{D}\llcorner k] \\
\overline{\texttt{\{}C\texttt{ l}_i\texttt{:\}}} &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, \delta(\vec{l}).\, \overline{C}\llcorner(\mathsf{D}\llcorner l_i)\, s\llcorner[(\vec{l}) \mid i\colon k] \\
\mathsf{D} &\stackrel{\text{def}}{=} \mathsf{Y}\, \lambda x.\, x : K \multimap K \\
\vec{l} &\stackrel{\text{def}}{=} l_0, \ldots, l_{n-1}
\end{aligned}$$

Programs

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{S} \to \mathbf{R}$$

$$\begin{aligned}
\llcorner C \lrcorner &\stackrel{\text{def}}{=} \delta t.\, \lambda s.\, \overline{C}\llcorner(\lambda s.\, \delta(\vec{l}).\, t\, s)\, s\llcorner(\{\mathsf{D}\}_0^{n-1}) \\
\mathsf{D} &\stackrel{\text{def}}{=} \mathsf{Y}\, (\lambda y.\, \lambda x.\, x\llcorner(y\, x))\, \delta k.\, k : K
\end{aligned}$$

10.8 Semantic Landscape Revisited

Having presented the details of stored labels, we now summarize the semantic landscape we have been exploring and position the developments in the last few chapters as well as the semantics we considered in this chapter. As indicated in Section 10.1, we considered three basic sorts of semantics:

10.8.1 Control environment

First, interpretations of form (10.1), of which the first we presented were the semantics of the languages of Chapter 7 and Chapter 8. In both of these languages there is no way for a label to escape from (outlive) the label environment in which it is bound, so there is no need to store anything related to control. Technically, we can observe this by noting that in the refined interpretations of these languages

$$\&\, K \,\&\, K \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mathbf{S} \to \mathbf{R} \tag{10.5}$$

the store contains only numbers, which are neither label L-values (which are target language identifiers) nor label R-values (which are continuations). Hence there is no connection between the store and the label assignment.

In the present chapter we connected the store and the label assignment first with an interpretation of the form

$$(L \to K) \,\&\, K \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} (\mathbf{N} \to V) \to \mathbf{R}$$

Unlike in (10.5), where there was no interaction between label L-values, L, and stored values, V, here we store label L-values. That is, we wanted a situation something like $V = \mathbf{N} \cup L$, but to make this simple, we took $L = \mathbf{N}$ and used $\mathbf{N} \cup \mathbf{N} = \mathbf{N}$, yielding

$$(\mathbf{N} \to K) \,\&\, K \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mathbf{S} \to \mathbf{R}$$

We used this semantics to interpret two source languages: first one with stored global labels in Section 10.2, and then one with dynamically-assigned (or fluidly-bound) labels in Section 10.4. While stored global labels are an instance of dynamically-assigned labels, we presented them anyhow since dynamic-assignment is not as widely understood and the interpretation of stored global labels can illustrate the types used in the semantics without worrying about strange binding forms. On the other hand, we presented dynamically-assigned labels since stored global labels are a rather weak and degenerate control construct which does not come close to fully exercising the types in the semantics. The interpretation of dynamically-assigned labels also serves to demonstrate that stored nested labels are not inherently incompatible with restricted typing—as Section 10.3 might lead one to believe—if care is taken not to store control.

10.8.2 Code store

In Section 10.5, we considered a semantics with a code store:

$$K \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mu K.\, \mathbf{S} \to (\mathbf{N} \to K \multimap K) \to \mathbf{R}$$

Here we see that the code store maps locations, $\mathbf{N}$, to code, or commands, $K \multimap K$. The code store need not be used linearly since it is part of the data context, not part of the control context as the label assignment is in semantics like (10.2). So, technically, this case is easy since refining the interpretation does not impose any significant constraints.

Nevertheless, this case fills an important conceptual place, highlighting the distinction between stored code and stored control. The basic point is that some finite amount of code (such as a command) is just data, but some finite code together with information determining what code will execute afterward is a control point, and hence is part of a control context.

10.8.3 Control store

We now turn to stored control and interpretations of form (10.2). Unlike the benign nature of storing code with respect to linear use of control contexts, storing control is highly malignant. Generally speaking, once something is dumped into any sort of general store, any hope of stylized usage is gone.

But stored control does not entirely preclude restricted use of control contexts: the language of coroutines in Chapter 9 provides stored control in the form of the control state of the blocked coroutine. Note that the control state of a coroutine is an instance of stored control and not merely stored code because the blocked coroutine, once unblocked, need not ever relinquish control, and hence a coroutine's control state must determine an entire future computation, not just some code. In this case the control store is implicit in the source language, and is manipulated in a very stylized way, making possible a refined interpretation (of form (10.2)):

$$K \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mu K.\, \mathbf{S} \to K \multimap \mathbf{R}$$

Here, while there is nothing control-related in the data store, the label assignment acts as a separate control store. The upward nature of continuations is also clear here as we explicitly have continuation arguments to continuations.

In Section 10.3, we considered a semantics of form

$$\mathbin{\&} K \mathbin{\&} K \to K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mu K.\, \mathbf{S} \to (\mathbf{N} \to K) \to \mathbf{R}$$

where we have not only a control environment mapping labels to continuations, but also a control store mapping locations, $\mathbf{N}$, to continuations. Despite the presence of a control environment, this semantics is like (10.2) in the crucial aspect that it manipulates a control store. We used this semantics to interpret a language with (statically-bound) stored nested labels. As we noted in Section 7.1, the forward-jumping blocks we use are a sort of first-order analogue of `call/cc`, and now with the ability to store labels, labels (and hence, continuations) are upward and can outlive the blocks which defined them, resulting in full stored control. Since this store is essentially unconstrained, attempting to give a refined interpretation fails.

To better understand what a linear interpretation of stored control requires of a source language, we presented, in Section 10.6, a language with explicit control store in the form of mutable labels and then, in Section 10.7, modified this language, in an admittedly unnatural way, to allow an affine semantics. The natural but unrefined interpretation fits form (10.2):

$$K \to K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mu K.\, \mathbf{S} \to \mathbin{\&} K \to \mathbf{R}$$

and the refined interpretation has similar form:

$$K \multimap K \quad \text{where} \quad K \stackrel{\text{def}}{=} \mu K.\, \mathbf{S} \to \bigotimes K \multimap \mathbf{R}$$

The result of the restrictions allowing the refined interpretation is that all the operations on the control store must be very careful to move, not copy, stored pieces of control context. Unsurprisingly, since the programming abstraction (the control store) is entirely unidiomatic, the programmer is left with all the work. In other words, while we use an untyped and unsafe language, if we were to use a typed and safe one we would be forced to pull linearity from the target type system into the source type system as well, or something equivalent.

10.9 Appendix: Target Language $\left(+= (\cdot) \otimes (\cdot)\right)$

In this section we present the technical details for the multiplicative product, $(\cdot) \otimes (\cdot)$, we use in this chapter in addition to the additive product, $(\cdot) \mathbin{\&} (\cdot)$. All this is conceptually summarized by the typing rule for constructing a $\otimes$-pair:

$$[\text{MPAIR}]\ \frac{\Gamma\ ;\ \Delta \vdash M : P \quad \Gamma\ ;\ \Delta' \vdash N : Q}{\Gamma\ ;\ \Delta\ ,\ \Delta' \vdash (M,N) : P \otimes Q}$$

This indicates that the factors of a $\otimes$-pair do not share the same restricted zone, as the factors of a $\&$-pair do, but must instead depend on disjoint resources.

10.9.1 Syntax

We extend the grammar of terms of Section 2.2:

$M ::=$	$\cdots$	*terms*
	$\mid (M, M)$	multiplicative pair
	$\mid \delta(x,x).\, M$	multiplicative pattern match

And, in Section A.1 we define the following syntactic sugar:

$\mid ()$	multiplicative unit constant
$\mid (M)$	unary $\otimes$-tuple
$\mid \delta(x,\ldots,x).\, M$	n-ary restricted multiplicative pattern match
$\mid [(M,\ldots,M) \mid i{:}\, M]$	n-ary $\otimes$-tuple extension

Convention: $\otimes$-pairing is right-associative, so (M,N,O) parses to $(M,(N,O))$ rather than $((M,N),O)$. The body of a multiplicative pattern match extends as far to the right as possible, so $\delta(x,y).\, M\, N$ parses to $\delta(x,y).\, (M\, N)$ rather than $(\delta(x,y).\, M)\, N$.

10.9.2 Equational theory

We add the following to the axioms of Section 3.2:

$$\overline{(\delta(x_0,x_1).\, M)\_(M_0,M_1) \stackrel{\beta\eta}{=} M[x_0,x_1 \mapsto M_0,M_1]}$$

10.9.3 Type system

The grammar of types of Section 3.2 is extended by

$P ::=$	$\cdots$	*pointed types*
	$\mid P \otimes P$	multiplicative product type

And, in Section A.3 we define the following syntactic sugar:

$\mid \bigotimes_n P$	n-ary multiplicative product type

Convention: The multiplicative product type constructor is right-associative, so $P \otimes Q \otimes R$ parses to $P \otimes (Q \otimes R)$ rather than $(P \otimes Q) \otimes R$. The precedence of $(\cdot) \otimes (\cdot)$ is the same as $(\cdot)\ \&\ (\cdot)$ and higher than $(\cdot) \to (\cdot)$ and $(\cdot) \multimap (\cdot)$, so $P \otimes Q \multimap R$ parses to $(P \otimes Q) \multimap R$ rather than $P \otimes (Q \multimap R)$. The n-ary multiplicative product type constructor has higher precedence

than that of all the binary type constructors, so $\bigotimes_n A \multimap B$ parses to $(\bigotimes_n A) \multimap B$ rather than $\bigotimes_n (A \multimap B)$ and $\bigotimes_n A \otimes B$ parses to $(\bigotimes_n A) \otimes B$ rather than $\bigotimes_n (A \otimes B)$. When it is clear from context, we generally omit n and write $\bigotimes P$ for $\bigotimes_n P$.

The typing rules of Section 3.2 are extended with

$$[\textsc{MPair}]\ \frac{\Gamma\ ;\ \Delta \vdash M : P \quad \Gamma\ ;\ \Delta' \vdash N : Q}{\Gamma\ ;\ \Delta\ ,\ \Delta' \vdash (M,N) : P \otimes Q} \qquad [\textsc{MPatM}]\ \frac{\Gamma\ ;\ \Delta\ ,\ x_0 : P_0\ ,\ x_1 : P_1 \vdash M : Q}{\Gamma\ ;\ \Delta \vdash \delta(x_0,x_1).\,M : P_0 \otimes P_1 \multimap Q}$$

The point to note about the multiplicative product is that, unlike $(\cdot)\ \&\ (\cdot)$, the restricted zone is not shared by the factors, instead they must depend on separate restricted identifiers.

Chapter 11

Everything at Once

In previous chapters we always interpreted a single control construct in isolation. While this simplified the various individual semantics, it might leave the impression, or cause the anxiety, that the simplicity of the surrounding language was crucial to each control construct's interpretation. In this chapter we counter by collecting many of the various control constructs previously considered into one conglomerate language. The refined interpretation of this language follows straightforwardly from the interpretations of the individual control constructs given in previous chapters. In this regard, there are no new concepts. On the other hand, this language demonstrates a wide variety of control constructs which simultaneously admit an interpretation where the use of control contexts is restricted. And the straightforwardness of the interpretation, given the previous interpretations, offers some confidence that the individual interpretations are not artificially exploiting degeneracies of the particular simple languages considered.

11.1 Source Language

For the source language we start with the simple command language of Chapter 6 and extend it in a number of steps. First we add a form of (parameterless) procedure call and return (`valof` and `resultis`) inspired by Strachey and Wadsworth's original [SW74]. Next we add nested blocks of dynamically-assigned labels, as in Section 10.4, but we now allow backward as well as forward jumps to them, as in Chapter 8. Next we add coroutines, as in Chapter 9, and finally add stored commands as in Section 10.5. So this language encompasses essentially all the different control behaviors we have studied: procedures (though parameterless), exceptions (recursive dynamically-assigned labels provide the necessary control machinery), labels (recursive dynamically-assigned labels is the "most expressive" variety of control environment we studied which admits a refined interpretation), coroutines (providing natural and implicit control store), and stored commands (providing code store). Note that, despite its appearance, this language is higher-order; In particular, its semantics requires a reflexive domain equation, just like untyped λ-calculus.

The syntax is given by the grammar

E ::=		*expressions*
	$\mid n$	numeric literal
	$\mid *n$	dereference
	$\mid E$ == E	numeric equality
	$\mid$ `valof` C	valof (call)

	$\mid \texttt{l}_i$	label	
$C ::=$		*commands*	
	$\mid n = E$	assignment	
	$\mid \texttt{skip}$	no-op	
	$\mid C ; C$	sequence	
	$\mid \texttt{if(}E\texttt{)\{}C\texttt{\}}$	conditional	
	$\mid \texttt{resultis}\, E$	resultis (return)	
	$\mid \texttt{goto}\, E$	jump	
	$\mid \{C ; \{\texttt{l}_i : C\}_{i=1}^{n}\}$	block	$0 \leq n$
	$\mid \texttt{swap}$	swap coroutines	
	$\mid n = (C)$	store command	
	$\mid \texttt{exec} * n$	execute stored command	
$P ::=$		*programs*	
	$\mid C \mid C$	parallel	

where the n and i are numbers.

Except for a few points, the intended semantics follows from previous chapters. For a block, the scope of all the labels is all the commands, which allows backward jumps. Evaluation of `valof` C causes command C to be executed until it executes a `resultis` E command, which causes the original `valof` C expression to evaluate to the value of E. If execution of C finishes without executing a `resultis`, then `valof` C evaluates to 0.[1] Regarding the interaction between `valof`/`resultis` and stored commands; when a stored command is run and executes a `resultis` command, it returns to the dynamically enclosing `valof` expression (as a procedure returns to the dynamically enclosing caller) rather than the statically (lexically) enclosing `valof` expression (as a first-class continuation captured with `call/cc` refers to its statically current continuation).

It is an error to jump to an unbound label or to return a value with `resultis` outside of a `valof` expression.

11.2 Refined CPS Transformation

We present the interpretation of this language in several steps, and summarize the whole semantics in Figure 11.1.

11.2.1 Simple command language + `valof` and `resultis`

We start with the interpretation presented in Section 6.4 and extend it to handle `valof` and `resultis`.

Due to the addition of `valof`, expressions are no longer pure with respect to control, and so we give a CPS semantics of them instead of a direct semantics as we have done previously. This has essentially made expressions into just another sort of commands, but Strachey and Wadsworth's original intent was that this language's semantics would have to deal with "inconvenient" features.

As in Section 6.4, the type of command continuations is

$$K \stackrel{\text{def}}{=} \mathbf{S} \to \mathbf{R}$$

[1] This is different from Strachey and Wadsworth, who specify failure in this situation. We differ just to avoid carrying yet another continuation around in the semantics.

Expressions are interpreted much like procedures in Section 2.3 and Section 3.3: with a type of continuation transformers

$$\underbrace{(\mathbf{N} \to K)}_{\substack{\text{return}\\ \text{continuation}}} \multimap \underbrace{K}_{\substack{\text{call}\\ \text{continuation}}}$$

The asymmetry between call and return continuations here is due to the fact that `valof`/`resultis` provides a sort of parameterless procedures, so the call continuation accepts only a state, while the return continuation accepts both a value and a state. Likewise, commands will need access to a return continuation, so commands are interpreted with the type

$$\underbrace{(\mathbf{N} \to K)}_{\substack{\text{return}\\ \text{continuation}}} \mathbin{\&} \underbrace{K}_{\substack{\text{current}\\ \text{continuation}}} \multimap K$$

The transformations of expressions and commands are mutually recursive. The transformation of expressions is

$$\begin{aligned}
\overline{n} &\stackrel{\text{def}}{=} \delta r.\, r\, n \\
\overline{\texttt{*}n} &\stackrel{\text{def}}{=} \delta r.\, \lambda s.\, r\, s[n]\, s \\
\overline{E_0 \texttt{ == } E_1} &\stackrel{\text{def}}{=} \delta r.\, \overline{E_0} \llcorner \lambda e_0.\, \overline{E_1} \llcorner \lambda e_1.\, r\, (e_0 \stackrel{n}{=} e_1) \\
\overline{\texttt{valof}\, C} &\stackrel{\text{def}}{=} \delta r.\, \overline{C} \llcorner \langle r, r\, 0 \rangle
\end{aligned}$$

The effect of evaluating a `valof` C expression is to execute command C with the expression's return continuation as C's return continuation, and a current continuation should C fall of its end without returning with `resultis`. Note that rather than specifying failure in this case, our semantics treats such commands as if they had returned 0. Making them fail would be straightforward, however.

The transformation of the commands relevant at this step is

$$\begin{aligned}
\overline{n \texttt{ = } E} &\stackrel{\text{def}}{=} \delta \langle r, k \rangle.\, \overline{E} \llcorner \lambda e.\, \lambda s.\, k\, [s \mid n\texttt{:}\, e] \\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta \langle r, k \rangle.\, k \\
\overline{C_0 \texttt{;}\, C_1} &\stackrel{\text{def}}{=} \delta \langle r, k \rangle.\, \overline{C_0} \llcorner \langle r, \overline{C_1} \llcorner \langle r, k \rangle \rangle \\
\overline{\texttt{if(}E\texttt{) \{}C\texttt{\}}} &\stackrel{\text{def}}{=} \delta \langle r, k \rangle.\, \overline{E} \llcorner \lambda e.\, e \to k \,[]\, \overline{C} \llcorner \langle r, k \rangle \\
\overline{\texttt{resultis}\, E} &\stackrel{\text{def}}{=} \delta \langle r, k \rangle.\, \overline{E} \llcorner r
\end{aligned}$$

Here we see that the effect of executing a `resultis` E command is simply to evaluate (that is, execute) the expression E with the argument return continuation. This will then cause the dynamically-enclosing `valof` expression to evaluate to the value of E.

For now, we only consider programs consisting of a single command, rather than two in parallel, and interpret them with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

by

$$\lfloor C \rfloor \stackrel{\text{def}}{=} \delta t.\, \overline{C} \llcorner \langle \mathsf{D}, t \rangle$$

where we use the abbreviation

$$\mathsf{D} \stackrel{\text{def}}{=} \lambda n.\, (\mathsf{Y}\, \lambda x.\, x) \llcorner t : \mathbf{N} \to K$$

Here we see that returning outside of a `valof` expression, an error in the source language, causes execution to diverge.

11.2.2 Adding recursive dynamically-assigned labels

Following Section 10.4.2, with the addition of nested blocks of labels, a label environment is necessary. So commands are now interpreted with the type

$$\underbrace{(\mathbf{N} \to K)}_{\substack{\text{continuation}\\ \text{environment}}} \;\&\; \underbrace{(\mathbf{N} \to K)}_{\substack{\text{return}\\ \text{continuation}}} \;\&\; \underbrace{K}_{\substack{\text{current}\\ \text{continuation}}} \;\multimap K$$

Since expressions can contain commands, they also need a label environment, and so are now interpreted with the type

$$\underbrace{(\mathbf{N} \to K)}_{\substack{\text{continuation}\\ \text{environment}}} \;\&\; \underbrace{(\mathbf{N} \to K)}_{\substack{\text{return}\\ \text{continuation}}} \;\multimap K$$

The transformation of expressions is

$$\begin{aligned}
\overline{n} &\stackrel{\text{def}}{=} \delta\langle u, r\rangle.\, r\, n \\
\overline{*n} &\stackrel{\text{def}}{=} \delta\langle u, r\rangle.\, \lambda s.\, r\, s[n]\, s \\
\overline{E_0 \texttt{ == } E_1} &\stackrel{\text{def}}{=} \delta\langle u, r\rangle.\, \overline{E_0} \llcorner \langle u, \lambda e_0.\, \overline{E_1} \llcorner \langle u, \lambda e_1.\, r\, (e_0 \stackrel{n}{=} e_1)\rangle\rangle \\
\overline{\texttt{valof}\, C} &\stackrel{\text{def}}{=} \delta\langle u, r\rangle.\, \overline{C} \llcorner \langle u, r, r\, 0\rangle \\
\overline{\texttt{l}_i} &\stackrel{\text{def}}{=} \delta\langle u, r\rangle.\, r\, i
\end{aligned}$$

This extension is straightforward but for the interpretation of labels, which is the CPS version of their interpretation in Section 10.2.2.

Extending the transformation of the old commands is straightforward:

$$\begin{aligned}
\overline{n \texttt{ = } E} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \overline{E} \llcorner \langle u, \lambda e.\, \lambda s.\, k\, [s \mid n\texttt{:}\, e]\rangle \\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, k \\
\overline{C_0 \texttt{;}\, C_1} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \overline{C_0} \llcorner \langle u, r, \overline{C_1} \llcorner \langle u, r, k\rangle\rangle \\
\overline{\texttt{if(}E\texttt{) \{}C\texttt{\}}} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \overline{E} \llcorner \langle u, \lambda e.\, e \to k \,[]\, \overline{C} \llcorner \langle u, r, k\rangle\rangle \\
\overline{\texttt{resultis}\, E} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \overline{E} \llcorner \langle u, r\rangle
\end{aligned}$$

For the new commands, we follow Section 10.4.2 but extend to multi-label blocks roughly following Section 10.2.2, and treat recursion following Section 8.2, leading to an affine transformation:

$$\begin{aligned}
\overline{\texttt{goto}\, E} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \overline{E} \llcorner \langle u, \lambda e.\, u[e]\rangle \\
\overline{\{C_0 \texttt{;}\, \{\texttt{l}_i\texttt{:}\, C_i\}_{i=1}^{n}\texttt{\}}} &\stackrel{\text{def}}{=} \pi_0 \llcorner (\mathsf{Y}^\circ\, \delta\langle \{x_i\}_{i=0}^{n}\rangle.\, \langle \{\delta\langle u, r, k\rangle.\, \overline{C_i} \llcorner \langle \mathsf{U}, r, x_{i+1} \llcorner \langle u, r, k\rangle\rangle\}_{i=0}^{n-1} \\
&\qquad\qquad , \delta\langle u, r, k\rangle.\, \overline{C_n} \llcorner \langle \mathsf{U}, r, k\rangle\rangle)
\end{aligned}$$

where

$$\mathsf{U} \stackrel{\text{def}}{=} [\cdots [u \mid 1 : x_1 \llcorner \langle u, r, k \rangle] \cdots \mid n : x_n \llcorner \langle u, r, k \rangle] : \mathbf{N} \to K$$

For `goto` E, E is evaluated and the value bound to e. The continuation denoting `goto` E is then the continuation the environment, u, maps e to, $u[e]$. In this language, blocks bind multiple labels which can be jumped backwards to, and hence, all the labels in a block are mutually recursive. To interpret this, the transformation builds a recursively defined tuple of the interpretations of all the commands in the block, and projects out the first as the denotation of the block. This recursive tuple is defined as the least fixed-point of a function from such tuples to such tuples. Each element of the output tuple of this function is the denotation of a command, and follows the semantics of sequencing, but instead of the current continuation being defined in terms of the transformation of the second command, it is defined in terms of the corresponding element of the input tuple. The current continuation of the final command in the block is the current continuation of the block as a whole. The label environment for all the commands in the block, U, is the block's label environment, u, extended with bindings for all the labels bound by the block, $\{1, \ldots, n\}$. Each label lvalue i is bound to the ith factor of the recursive tuple of command meanings applied to the block's label environment and continuations, just as for the current continuations of the denotations of commands.

We still only consider programs consisting of a single command, rather than two in parallel, and interpret them with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel} \\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

by

$$\llcorner C \lrcorner \stackrel{\text{def}}{=} \delta t . \overline{C} \llcorner \langle \mathsf{D}, \mathsf{D}, t \rangle$$

where we use the abbreviation

$$\mathsf{D} \stackrel{\text{def}}{=} \lambda n . (\mathsf{Y}\, \lambda x . x) \llcorner t : \mathbf{N} \to K$$

Here we see that in addition to returning outside of a `valof` expression, jumping to an unbound label, again an error in the source language, causes execution to diverge.

11.2.3 Adding coroutines

We now give an affine delimited continuation interpretation, following Section 9.2.2, to handle coroutines. Rather than command continuations, we now use a type of command delimited continuations

$$K = \mathbf{S} \to (\mathbf{S} \to \mathbf{R}) \multimap \mathbf{R}$$

and add a control store for the control state of the blocked coroutine:

$$K \stackrel{\text{def}}{=} \mu K . \mathbf{S} \to K \multimap (\mathbf{S} \to K \multimap \mathbf{R}) \multimap \mathbf{R}$$

The interpretation of expressions remains unchanged, and commands are still interpreted with the type

$$(\mathbf{N} \to K) \,\&\, (\mathbf{N} \to K) \,\&\, K \multimap K$$

but now unrolling the recursive type on the right-hand side of the $(\cdot) \multimap (\cdot)$ yields

$$\underbrace{(\mathbf{N} \to K)}_{\substack{\text{label} \\ \text{environment}}} \,\&\, \underbrace{(\mathbf{N} \to K)}_{\substack{\text{return} \\ \text{continuation}}} \,\&\, \underbrace{K}_{\substack{\text{running} \\ \text{coroutine}}} \multimap \mathbf{S} \to \underbrace{K}_{\substack{\text{blocked} \\ \text{coroutine}}} \multimap \underbrace{(\mathbf{S} \to K \multimap \mathbf{R})}_{\substack{\text{shared} \\ \text{toplevel}}} \multimap \mathbf{R}$$

As usual when simply refining the result type, we need only add a transformation clause for the command(s) which manipulate the newly revealed type component:

$$\overline{\texttt{swap}} \stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\ \lambda s.\ \delta b.\ b\, s\, k$$

But for carrying the label environment and return continuation around, this interpretation is unchanged from that in Section 9.2.2.

Programs are interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel}\\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

by

$$\llcorner C_0 \mid C_1 \lrcorner \stackrel{\text{def}}{=} \delta t.\ \lambda s.\ \overline{C_0}\, \langle \mathsf{D}, \mathsf{D}, \mathsf{E}\rangle\, s\, (\overline{C_1}\, \langle \mathsf{D}, \mathsf{D}, \mathsf{E}\rangle)\, \lambda s.\ \delta b.\ t\, s$$

where we use the abbreviations

$$\begin{aligned} \mathsf{D} &\stackrel{\text{def}}{=} \lambda n.\ \mathsf{Y}\, \lambda x.\ x \qquad : \mathbf{N} \to K \\ \mathsf{E} &\stackrel{\text{def}}{=} \lambda s.\ \delta b.\ \delta p.\ p\, s\, b : K \end{aligned}$$

The definition of D differs here because it no longer needs to consume the toplevel continuation, it is instead embedded into the shared toplevel delimited continuation, $\lambda s.\ \delta b.\ t\, s$. Again, the only difference from the interpretation in Section 9.2.2 is carrying the label environment and return continuation around.

11.2.4 Adding stored commands

We now extend the interpretation to handle stored commands, following Section 10.5.2. With a different type interpreting commands than in Section 10.5.2, the type of the code store needed to interpret stored commands changes accordingly:

$$\mathsf{T} \stackrel{\text{def}}{=} \mathbf{N} \to (\mathbf{N} \to K)\ \&\ (\mathbf{N} \to K)\ \&\ K \multimap K$$

With this, we refine the result type to include the control store, making the type of continuations

$$K \stackrel{\text{def}}{=} \mu K.\ \mathbf{S} \to K \multimap (\mathbf{S} \to K \multimap \mathsf{T} \to \mathbf{R}) \multimap \mathsf{T} \to \mathbf{R}$$

And so the type of commands is (unrolling K once)

$$\underbrace{(\mathbf{N} \to K)}_{\substack{\text{label}\\ \text{env.}}}\ \&\ \underbrace{(\mathbf{N} \to K)}_{\substack{\text{return}\\ \text{cont.}}}\ \&\ \underbrace{K}_{\substack{\text{running}\\ \text{coroutine}}} \multimap \mathbf{S} \to \underbrace{K}_{\substack{\text{blocked}\\ \text{coroutine}}} \multimap \underbrace{(\mathbf{S} \to K \multimap \mathsf{T} \to \mathbf{R})}_{\substack{\text{shared}\\ \text{toplevel}}} \multimap \underbrace{\mathsf{T}}_{\substack{\text{code}\\ \text{store}}} \to \mathbf{R}$$

The transformation clauses for the additional commands are[2]

$$\begin{aligned} \overline{n = (C)} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\ \lambda s.\ \delta b.\ \delta p.\ \lambda t.\ k\, s\, b\, p\, [t \mid n{:}\overline{C}] \\ \overline{\texttt{exec}\ {*}n} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\ \lambda s.\ \delta b.\ \delta p.\ \lambda t.\ t[n]\, \langle u, r, k\rangle\, s\, b\, p\, t \end{aligned}$$

[2] At times like this one wishes for a labeled λ-calculus, which passes arguments by "label" rather than "position". This would allow a form of η-contraction for arguments to curried functions other than the last, such as s, b, p in the clause for $\overline{n = (C)}$.

Conceptually, these clauses are no different than in Section 10.5.2, there are simply more inert components to carry around.

Programs are still interpreted with the type

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\substack{\text{toplevel} \\ \text{continuation}}} \multimap \mathbf{S} \to \mathbf{R}$$

but now by the transformation

$$\lfloor C_0 \mid C_1 \rfloor \stackrel{\text{def}}{=} \delta k.\, \lambda s.\, \overline{C_0} \sqcup \langle \mathsf{D}, \mathsf{D}, \mathsf{E} \rangle\, s \sqcup (\overline{C_1} \sqcup \langle \mathsf{D}, \mathsf{D}, \mathsf{E} \rangle) \sqcup (\lambda s.\, \delta b.\, \lambda t.\, k\, s)\, \mathsf{D}$$

where we use the abbreviations

$$\begin{aligned} \mathsf{D} &\stackrel{\text{def}}{=} \lambda n.\, \mathsf{Y}\, \lambda x.\, x \qquad : \mathbf{N} \to P \\ \mathsf{E} &\stackrel{\text{def}}{=} \lambda s.\, \delta b.\, \delta p.\, p\, s \sqcup b : K \end{aligned}$$

from which we see that the every location in the code store in initialized to a divergent command. Also, now the toplevel delimited continuation, $\lambda s.\, \delta b.\, \lambda t.\, k\, s$, also throws away the code store before returning to the OS.

11.3 Conclusion

In this chapter we showed that the previous analyses of various control behaviors in isolation were not relying on that isolation by giving an interpretation of a language including virtually every control behavior we have considered: procedures (though parameterless), exceptions (recursive dynamically-assigned labels provide the necessary control machinery), labels (recursive dynamically-assigned labels is the "most expressive" variety of control environment we studied which admits a refined interpretation), coroutines (providing natural and implicit control store), and stored commands (providing code store).

In addition to being possible, interpreting the conglomerate language follows straightforwardly from the individual interpretations, and is largely a matter of unioning the individual interpretations' &-tuples and refining the result type.

Figure 11.1 Refined CPS of Everything at Once

Expressions

$$\underbrace{(\mathbf{N} \to K)}_{\text{labels}} \,\&\, \underbrace{(\mathbf{N} \to K)}_{\text{return}} \multimap K$$

$$\begin{aligned}
\overline{n} &\stackrel{\text{def}}{=} \delta\langle u, r\rangle.\, r\, n \\
\overline{*n} &\stackrel{\text{def}}{=} \delta\langle u, r\rangle.\, \lambda s.\, r\, s[n]\, s \\
\overline{E_0 \texttt{ == } E_1} &\stackrel{\text{def}}{=} \delta\langle u, r\rangle.\, \overline{E_0} \lrcorner \langle u, \lambda e_0.\, \overline{E_1} \lrcorner \langle u, \lambda e_1.\, r\, (e_0 \stackrel{n}{=} e_1)\rangle\rangle \\
\overline{\texttt{valof}\, C} &\stackrel{\text{def}}{=} \delta\langle u, r\rangle.\, \overline{C} \lrcorner \langle u, r, r\, 0\rangle \\
\overline{\texttt{l}_i} &\stackrel{\text{def}}{=} \delta\langle u, r\rangle.\, r\, i
\end{aligned}$$

Commands

$$(\mathbf{N} \to K) \,\&\, (\mathbf{N} \to K) \,\&\, K \multimap K$$

$$\begin{aligned}
K &\stackrel{\text{def}}{=} \mu K.\, \mathbf{S} \to K \multimap (\mathbf{S} \to K \multimap \mathsf{T} \to \mathbf{R}) \multimap \mathsf{T} \to \mathbf{R} \\
\mathsf{T} &\stackrel{\text{def}}{=} \mathbf{N} \to (\mathbf{N} \to K) \,\&\, (\mathbf{N} \to K) \,\&\, K \multimap K
\end{aligned}$$

$$\begin{aligned}
\overline{n \texttt{ = } E} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \overline{E} \lrcorner \langle u, \lambda e.\, \lambda s.\, k\, [s \mid n\texttt{:}\, e]\rangle \\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, k \\
\overline{C_0 \texttt{ ; } C_1} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \overline{C_0} \lrcorner \langle u, r, \overline{C_1} \lrcorner \langle u, r, k\rangle\rangle \\
\overline{\texttt{if(}E\texttt{) \{}C\texttt{\}}} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \overline{E} \lrcorner \langle u, \lambda e.\, e \to k \,[]\, \overline{C} \lrcorner \langle u, r, k\rangle\rangle \\
\overline{\texttt{resultis}\, E} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \overline{E} \lrcorner \langle u, r\rangle \\
\overline{\texttt{goto}\, E} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \overline{E} \lrcorner \langle u, \lambda e.\, u[e]\rangle \\
\overline{\texttt{\{}C_0\texttt{ ; \{l}_i\texttt{: }C_i\texttt{\}}_{i=1}^{n}\texttt{\}}} &\stackrel{\text{def}}{=} \pi_0 \lrcorner (\mathsf{Y}^\circ\, \delta\langle \{x_i\}_{i=0}^{n}\rangle.\, \langle \{\delta\langle u, r, k\rangle.\, \overline{C_i} \lrcorner \langle \mathsf{U}, r, x_{i+1} \lrcorner \langle u, r, k\rangle\rangle\}_{i=0}^{n-1} \\
&\qquad\qquad , \delta\langle u, r, k\rangle.\, \overline{C_n} \lrcorner \langle \mathsf{U}, r, k\rangle\rangle) \\
\overline{\texttt{swap}} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \lambda s.\, \delta b.\, b\, s \lrcorner k \\
\overline{n \texttt{ = (}C\texttt{)}} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \lambda s.\, \delta b.\, \delta p.\, \lambda t.\, k\, s \lrcorner b \lrcorner p\, [t \mid n\texttt{:}\, \overline{C}] \\
\overline{\texttt{exec }{*}n} &\stackrel{\text{def}}{=} \delta\langle u, r, k\rangle.\, \lambda s.\, \delta b.\, \delta p.\, \lambda t.\, t[n] \lrcorner \langle u, r, k\rangle\, s \lrcorner b \lrcorner p\, t \\
\mathsf{U} &\stackrel{\text{def}}{=} [\cdots [u \mid 1\texttt{:}\, x_1 \lrcorner \langle u, r, k\rangle] \cdots \mid n\texttt{:}\, x_n \lrcorner \langle u, r, k\rangle] : \mathbf{N} \to K
\end{aligned}$$

Programs

$$\underbrace{(\mathbf{S} \to \mathbf{R})}_{\text{toplevel}} \multimap \mathbf{S} \to \mathbf{R}$$

$$\begin{aligned}
\llcorner C_0 \texttt{ | } C_1 \lrcorner &\stackrel{\text{def}}{=} \delta k.\, \lambda s.\, \overline{C_0} \lrcorner \langle \mathsf{D}, \mathsf{D}, \mathsf{E}\rangle\, s \lrcorner (\overline{C_1} \lrcorner \langle \mathsf{D}, \mathsf{D}, \mathsf{E}\rangle) \lrcorner (\lambda s.\, \delta b.\, \lambda t.\, k\, s)\, \mathsf{D} \\
\mathsf{D} &\stackrel{\text{def}}{=} \lambda n.\, \mathsf{Y}\, \lambda x.\, x \qquad : \mathbf{N} \to P \\
\mathsf{E} &\stackrel{\text{def}}{=} \lambda s.\, \delta b.\, \delta p.\, p\, s \lrcorner b : K
\end{aligned}$$

Chapter 12

Loose Ends and Related Work

12.1 Loose Ends (aka Remaining Irritations)

12.1.1 Definition of control context

We have argued in Chapter 5 that it is crucial to get the control context right, however we do not have a *definition* of control context, not even a decent informal one. In particular, it is not possible to take any given type we have used to interpret a language and change a $\multimap$ to an $\to$ and readily see how the interpretation then violates the definition of linearly used control contexts. This is an exceedingly irritating situation. In our defense, the distinction between continuations and control contexts is not generally made in semantics work and, as far as we know, no one has a reasonably general (generalizing more than one instance, say) characterization of this distinction.

It is important to appreciate that the form of the control context is not a function of a source language's possible control behavior, but of the form of the source language's semantics. For instance, some high-level control constructs (such as `call/cc`) which do not admit a refined interpretation can be compiled to low-level languages (such as the language of stored global labels) which do admit a refined interpretation. The key is that when the environment is implemented (as a tree) rather than being implicit in the semantics, it changes from being part of the control context to part of the data context, and hence you gain restricted typecheckability.

Additionally, the treatment of recursion in the language of backward jumps (Section 8.2.2) indicates that there some subtlety involved. The need to worry about the form of the types of the subterms of the semantics is irritating. However, this situation is not entirely surprising since "linear use of control contexts" is enforced through the form of types rather than the linear type system itself. Perhaps this indicates some inappropriateness of the underlying type theory to our analysis, but it seems more likely a consequence of not having a formal definition of control context.

Danvy and Nielsen have some work [DN01, Dan04] which treats continuations and control contexts (in our terminology) intensionally, as abstract data types. In this setting, the two can be related and distinguished, primarily by analyzing the "apply" operation. Perhaps an adaptation of that work to an extensional setting, using function spaces, would provide purchase for definitions distinguishing continuations and control contexts. Or, perhaps, the analysis in this thesis needs to be adapted to an intentional setting.

12.1.2 Remember versus use, and linear use versus separation

We explained, in Chapter 1, that the observation that `call/cc` allowed a continuation to be invoked more than once led us to the notion of linearly used control contexts. In hindsight, however, it appears that the crucial restriction which linearity enforces is not that control contexts

are "used" (only) once, but that they are not "remembered" after their first invocation. Said differently, linear typing enforces what is basically a "unique pointer" property (slightly modified to account for the additive product), ensuring that no control contexts are aliased. So linearity restricts multiple uses by preventing any situation in which multiple use might occur, rather than preventing the multiple uses themselves. This may seem irrelevant, but an analogy can be drawn with techniques to control pointer aliasing, where both approaches have been attempted. In that setting, the currently developing direct constraint approach, which controls aliasing essentially by checking on dereference [Rey02]; is looking promisingly flexible and manageable compared to the indirect constraint approach, which controls aliasing by checking on reference creation [Hof00, Wad91, WM01]. (Although it must be added that the former approach may be less realizable as a type system, and the two approaches differ in goals.) This ensures that the question of whether linearity is the correct formal foundation for this sort of analysis remains open. It seems quite likely that better results are there to be had, especially as one moves closer to actual low-level implementations and properties of interest in that context, by starting over with the notion of separation in place of linear use.

12.1.3 Explicit versus implicit recursion

Regarding the definition of control context, a particularly irritating case is the difference between the interpretations of untyped λ-calculus and backward jumps. In these two semantics, the control contexts are different, but why? What is the difference between the source languages which leads to this difference in control contexts? I would argue that the cause lies in recursion in λ-calculus being "implicit," that is, available only through self-application; while in the language of backward jumps it is "explicit," that is, hardcoded directly into the language. But the distinction between implicit and explicit recursion does not seem to be one that is made in semantics. It seems hard to believe that our analysis could be the only one sensitive to this difference.

In Section 8.2.2 we argued for the choice of control context somewhat technically by appeal to refining the result type. But there is also a speculative intuitive justification for the control contexts being different. When relying on self-application in the source language, the recursive continuation transformers used to implement recursion are source language procedures, and hence are not part of the control context of the language semantics. When implementing recursion directly, however, the recursive delimited continuations do not correspond to anything in the source language, and hence are part of the control context of the language semantics.

So in general, the impact on the division between data and control contexts entailed by explicit versus implicit recursion must be considered. That is, in the explicit case, the structures implementing recursion become part of the control context, whereas in the implicit case, they are part of the data context. In more detail, the untyped λ-calculus does not provide an implementation of recursion, programs must instead roll their own, generally using self-application, and there are several different possible recursions (naive, properly tail-recursive, memoizing, etc.). So recursion is an aspect of the program, not of the source language, and hence the infrastructure implementing recursion is not contained in the control context of the semantics of the source language. On the other hand, a particular recursion is a definite feature of the particular source language, and so its implementation is part of the control context in the semantics of the source language.

12.2 Related Work

There are several lines of development concerned with restricting CPS or using substructural type systems. It is important to note that our approach is very different from Filinski's linear continuations [Fil92]. Since Filinski used a linear target language, he might have accounted for linearly used continuations as we have; but his call-by-value transformation has an additional $!(\cdot)$,

which essentially turns the principal $(\cdot) \multimap (\cdot)$ we use into $(\cdot) \to (\cdot)$ and lifts all restrictions.

Completeness of affine CPS, or no junk, is in our experience not well known, and is even often disbelieved initially. It must be said, though, that close familiarity with the details of Sabry and Felleisen's work [SF93] would perhaps make one expect the result. In any case, we hope that our results give further support for the significance of the Sabry-Felleisen CPS sublanguage. Certainly, the use of a domain or type equation to characterize the target provides an independent rationale for it; the CPS language can be obtained by starting with types, and then adopting DILL-style typing rules, rather than by designing the grammar for it directly.

Danvy has given a different CPS sublanguage and DS transformation [Dan92, Dan94]. One way his language differs from that of Sabry and Felleisen is that it is not closed under $\beta\eta$; he thus characterizes the range of CPS exactly, rather than up to equivalence.

In a different line, Polakow, Pfenning, and Yi have given an account and further analysis of Danvy's approach using typing rules from a linear λ-calculus that lacks Exchange [PP00, PY01]. Compared to the approach here, an important point is their use of ordered contexts to capture the property that, in their treatment, the arguments of auxiliary continuations introduced by the CPS transformation are used in a stack-like fashion, and that all these arguments are used before the current continuation. The approach presented here, however, cannot capture these properties since we make no distinction between arguments of the auxiliary continuations and arguments of the continuations corresponding to source procedures. Also, once this distinction is made, the inherent notion of order in their system plays a crucial role.

As mentioned earlier in the context of completeness, while our results are not fully general since we "carve out" a sublanguage of the target, Hasegawa later showed the analogous but fully general result for the simply-typed source language [Has02]. However, this proof exploits the absence of divergence (using long $\beta\eta$-normal forms) and does not appear to generalize to the untyped case where normalization arguments are insufficient and one has to consider observational equivalence rather than simply $\beta\eta$-equality. Additionally, Laird [Lai03] later generalized the no junk result and (game-) semantically proved full abstraction of the affine CPS transformation. He considered the transformation from untyped λ-calculus into a full (without restricted types) calculus, and used observational equivalence rather than $\beta\eta$-equality.

Thielecke [Thi03] has investigated a connection between result type polymorphism (abstractness) and linear continuation-passing using a control effect system on the source language.

Appendix A

Target Language

Since we develop the target language incrementally as needed through the chapters, it is convenient to present it in its entirety. We also give various technicalities here to avoid interrupting the preceding chapters.

A.1 Syntax

The syntax of terms is given by the grammar in Figure A.1, and extended with the syntactic sugar in Figure A.2. (Note that these definitions are possibly recursive.) We identify terms up to renaming of bound identifiers. The set of identifiers occurring free in a term, $\mathsf{fi}((\cdot))$, is defined in the usual way and a term is *closed* if it has no free identifiers. The usual capture-avoiding substitution is written $(\cdot)[(\cdot) \mapsto (\cdot)]$, and the simultaneous variant is $(\cdot)[(\cdot), \ldots, (\cdot) \mapsto (\cdot), \ldots, (\cdot)]$. We make the following syntactic conventions:

- The body of an abstraction (of any sort) extends as far to the right as possible, so $\lambda x.\, M\, N$ parses to $\lambda x.\, (M\, N)$ rather than $(\lambda x.\, M)\, N$.
- Application (of any sort) is left-associative, so $M\, N\, O$ parses to $(M\, N)\, O$ rather than $M\, (N\, O)$.
- Pairing (of any sort) is right-associative, so $\langle M, N, O\rangle$ parses to $\langle M, \langle N, O\rangle\rangle$ rather than $\langle\langle M, N\rangle, O\rangle$.
- Numeric equality has lower precedence than application, so $M\, N \stackrel{n}{=} O$ parses to $(M\, N) \stackrel{n}{=} O$ rather than $M\, (N \stackrel{n}{=} O)$.
- The conditional has lower precedence than numeric equality, so $M \stackrel{n}{=} N \to O \,[]\, P$ parses to $(M \stackrel{n}{=} N) \to O \,[]\, P$ rather than $M \stackrel{n}{=} (N \to O \,[]\, P)$.
- Lookup has higher precedence than application, so $M\, S[N]$ parses to $M\, (S[N])$ rather than $(M\, S)[N]$.

Figure A.1 Target Term Syntax

$M ::=$	*terms*	
$\mid n$	numeric literal	
$\mid M \stackrel{n}{=} M$	numeric equality	
$\mid M \to M \,[]\, M$	conditional (if zero)	
$\mid x$	identifier	
$\mid \lambda x.\, M$	abstraction	
$\mid M\, M$	application	
$\mid \delta x.\, M$	restricted abstraction	
$\mid M \_ M$	restricted application	
$\mid \langle M, M \rangle$	additive pair	
$\mid \pi_i$	additive projection	$i \in \{0, 1\}$
$\mid (M, M)$	multiplicative pair	
$\mid \delta(x, x).\, M$	multiplicative pattern match	
$\mid \mathsf{halt}$	terminate	
$\mid \langle\rangle$	unit constant	
$\mid \lambda\langle\rangle.\, M$	unit consumer	
$\mid \delta\langle\rangle.\, M$	restricted unit consumer	

Figure A.2 Target Term Syntactic Sugar

$M ::=$	$\cdots$	*terms*
$\mid$	$\langle M\rangle \stackrel{\text{def}}{=} M$	unary &-tuple
$\mid$	$\pi_0^m \stackrel{\text{def}}{=} \pi_0$ $\pi_1^2 \stackrel{\text{def}}{=} \pi_1$ $\pi_{j+1}^{m+1} \stackrel{\text{def}}{=} \delta x.\, \pi_j^m \llcorner (\pi_1 \llcorner x)$	n-ary &-projection
$\mid$	$\lambda\langle x_0,\ldots,x_{n-1}\rangle.\, M[x \mapsto \langle x_0,\ldots,x_{n-1}\rangle] \stackrel{\text{def}}{=} \lambda x.\, M[x_0,\ldots,x_{n-1} \mapsto \pi_0 \llcorner x,\ldots,\pi_{n-1}\llcorner x]$	n-ary &-patm
$\mid$	$\delta\langle x_0,\ldots,x_{n-1}\rangle.\, M[x \mapsto \langle x_0,\ldots,x_{n-1}\rangle] \stackrel{\text{def}}{=} \delta x.\, M[x_0,\ldots,x_{n-1} \mapsto \pi_0 \llcorner x,\ldots,\pi_{n-1}\llcorner x]$	n-ary restricted &-patm
$\mid$	$() \stackrel{\text{def}}{=} \langle\rangle$	$\otimes$-unit constant
$\mid$	$(M) \stackrel{\text{def}}{=} M$	unary $\otimes$-tuple
$\mid$	$\delta().\, M \stackrel{\text{def}}{=} \delta\langle\rangle.\, M$ $\delta(x_0).\, M \stackrel{\text{def}}{=} \delta x_0.\, M$ $\delta(x_0,\ldots,x_{n-1}).\, M \stackrel{\text{def}}{=} \delta(x_0,x).\, (\delta(x_1,\ldots,x_{n-1}).\, M)\llcorner x$	n-ary restricted $\otimes$-patm $x \notin \mathsf{fi}(M)$ and $n \geq 1$
$\mid$	$S[N] \stackrel{\text{def}}{=} S\,N$	lookup
$\mid$	$[S \mid N\colon M] \stackrel{\text{def}}{=} \lambda o.\, o \stackrel{n}{=} N \to S[o] \,[]\, M$	extension $o \notin \mathsf{fi}(S) \cup \mathsf{fi}(N) \cup \mathsf{fi}(M)$
$\mid$	$[\langle M_0,\ldots,M_{n-1}\rangle \mid i\colon M] \stackrel{\text{def}}{=} \langle M_0,\ldots,M_{i-1},M,M_{i+1},\ldots,M_{n-1}\rangle$	n-ary &-tuple extension $0 \leq i < n$
$\mid$	$[(M_0,\ldots,M_{n-1}) \mid i\colon M] \stackrel{\text{def}}{=} (M_0,\ldots,M_{i-1},M,M_{i+1},\ldots,M_{n-1})$	n-ary $\otimes$-tuple extension $0 \leq i < n$

A.2 Equational Theory

The equational theory is generated in the usual way from the axioms in Figure A.3 and extended with axioms for the syntactic sugar in Figure A.4, which are derived:

[1]: Immediate, if $n = 0$. Otherwise

$$
\begin{aligned}
&(\delta\langle x_0,\ldots,x_{n-1}\rangle.\,M)\llcorner\langle M_0,\ldots,M_{n-1}\rangle\\
&= (\delta x.\,M[x_0,\ldots,x_{n-1}\mapsto \pi_0\llcorner x,\ldots,\pi_{n-1}\llcorner x])\llcorner\langle M_0,\ldots,M_{n-1}\rangle\\
&\overset{\beta\eta}{=} M[x_0,\ldots,x_{n-1}\mapsto \pi_0\llcorner x,\ldots,\pi_{n-1}\llcorner x][x\mapsto\langle M_0,\ldots,M_{n-1}\rangle]\\
&= M[x_0,\ldots,x_{n-1}\mapsto \pi_0\llcorner\langle M_0,\ldots,M_{n-1}\rangle,\ldots,\pi_{n-1}\llcorner\langle M_0,\ldots,M_{n-1}\rangle]\\
&\overset{\beta\eta}{=} M[x_0,\ldots,x_{n-1}\mapsto M_0,\ldots,M_{n-1}]
\end{aligned}
$$

Figure A.3 Target Equational Theory Axioms

$$\frac{}{n \overset{n}{=} n \overset{\beta\eta}{=} 1} \qquad \frac{}{m \overset{n}{=} n \overset{\beta\eta}{=} 0}\ m \neq n$$

$$\frac{}{0 \to M \,[]\, N \overset{\beta\eta}{=} M} \qquad \frac{}{n \to M \,[]\, N \overset{\beta\eta}{=} N}\ n \neq 0 \qquad \frac{}{n \to M \,[]\, M \overset{\beta\eta}{=} M}$$

$$\frac{}{(\lambda x.\, M)\, N \overset{\beta\eta}{=} M[x \mapsto N]} \qquad \frac{}{\lambda x.\, M\, x \overset{\beta\eta}{=} M}\ x \notin \mathsf{fi}(M)$$

$$\frac{}{(\delta x.\, M) \llcorner N \overset{\beta\eta}{=} M[x \mapsto N]} \qquad \frac{}{\delta x.\, M \llcorner x \overset{\beta\eta}{=} M}\ x \notin \mathsf{fi}(M)$$

$$\frac{}{\pi_i \llcorner \langle M_0, M_1 \rangle \overset{\beta\eta}{=} M_i} \qquad \frac{}{\langle \pi_0 \llcorner M, \pi_1 \llcorner M \rangle \overset{\beta\eta}{=} M}$$

$$\frac{}{(\delta(x_0, x_1).\, M) \llcorner (M_0, M_1) \overset{\beta\eta}{=} M[x_0, x_1 \mapsto M_0, M_1]}$$

$$\frac{}{(\lambda\langle\rangle.\, M)\, \langle\rangle \overset{\beta\eta}{=} M} \qquad \frac{}{(\delta\langle\rangle.\, M) \llcorner \langle\rangle \overset{\beta\eta}{=} M}$$

Figure A.4 Target Equational Theory Axiom for Syntactic Sugar

$$[1]\ \frac{}{(\delta\langle x_0, \ldots, x_{n-1}\rangle.\, M) \llcorner \langle M_0, \ldots, M_{n-1}\rangle \overset{\beta\eta}{=} M[x_0, \ldots, x_{n-1} \mapsto M_0, \ldots, M_{n-1}]}$$

A.3 Type System

The syntax of types is given by the grammar in Figure A.5 and extended with the syntactic sugar in Figure A.6. We identify types up to renaming of bound type identifiers, and make the following syntactic conventions:

- The function type constructors (all sorts) have equal precedence and are right-associative, so $S \to T \to U$ parses to $S \to (T \to U)$ rather than $(S \to T) \to U$.
- As in an abstraction term, the body of a recursive type extends as far to the right as possible, so $\mu X.\, P \multimap Q$ parses to $\mu X.\, (P \multimap Q)$ rather than $(\mu X.\, P) \multimap Q$.
- The product type constructors (both sorts) are right-associative, so $P \,\&\, Q \,\&\, R$ parses to $P \,\&\, (Q \,\&\, R)$ rather than $(P \,\&\, Q) \,\&\, R$. The product type constructors have the same precedence, which is higher than that of the function type constructors, so $P \,\&\, Q \multimap R$ parses to $(P \,\&\, Q) \multimap R$ rather than $P \,\&\, (Q \multimap R)$.
- The continuation type constructor binds tightest of all, so $\neg P \otimes Q$ parses to $(\neg P) \otimes Q$ rather than $\neg(P \otimes Q)$.
- The n-ary product type constructors have the same precedence, which is higher than that of all the binary type constructors, so $\&_n A \to B$ parses to $(\&_n A) \to B$ rather than $\&_n (A \to B)$ and $\&_n A \,\&\, B$ parses to $(\&_n A) \,\&\, B$ rather than $\&_n (A \,\&\, B)$.

Figure A.5 Target Type Syntax

$P ::=$		*pointed types*
	$\mid A \to P$	function type
	$\mid P \multimap P$	restricted function type
	$\mid P \mathbin{\&} P$	additive product type
	$\mid P \otimes P$	multiplicative product type
	$\mid \mathbf{R}$	result type
	$\mid \mathbf{1}$	unit type
	$\mid X$	type identifier
	$\mid \mu X.\, P$	recursive type
$A ::=$		*types*
	$\mid \mathbf{N}$	base type
	$\mid A \to A$	nonrecursive function type
	$\mid P$	pointed type

Figure A.6 Target Type Syntactic Sugar

$P ::=$	$\cdots$		*pointed types*
$\mid$	$\neg T \stackrel{\text{def}}{=} T \to \mathbf{R}$		continuation type
$\mid$	$\mathbin{\&}_n P \stackrel{\text{def}}{=} \begin{cases} \mathbf{1} & \text{if } n = 0 \\ \underbrace{P \mathbin{\&} \cdots \mathbin{\&} P}_{n\text{-many}} & \text{if } n \geq 1 \end{cases}$		n-ary additive product type
$\mid$	$\bigotimes_n P \stackrel{\text{def}}{=} \begin{cases} \mathbf{1} & \text{if } n = 0 \\ \underbrace{P \otimes \cdots \otimes P}_{n\text{-many}} & \text{if } n \geq 1 \end{cases}$		n-ary multiplicative product type

A *typing* $x : T$ annotates identifier x, called the *subject* of the typing, with its type T. A *context* $\Gamma \,;\, \Delta$ consists of an *unrestricted zone* Γ, which is a finite set of typings of the form $x : A$, and a *restricted zone* Δ, which is a finite set of typings of the form $x : P$, such that no identifier is the subject of more than one typing. The set of subjects of a context is denoted $\mathsf{sub}((\cdot))$. We use the notation $\Gamma \,,\, \Gamma'$ for the union of Γ and Γ' if the sets of subjects of the typings in Γ and Γ' are disjoint, otherwise $\Gamma \,,\, \Gamma'$ is undefined. Also, we use the notation $-$ for the empty set and implicitly coerce single typings into singleton sets.

The judgment $\Gamma \,;\, \Delta \vdash M : A$ states that M is a well-typed term of type A in context $\Gamma \,;\, \Delta$, and is defined by the rules in Figure A.7. (There is an implicit requirement that a judgment is derivable only if all the contexts involved in the derivation are well-defined. We leave this, and often other such requirements, unstated.) Contraction and Weakening in the unrestricted zone are built into the system and we will often implicitly contract or weaken unrestricted zones. Also, note that since contexts are built from sets (which are unordered), the Exchange rules are built into the system. These admissible rules are summarized in Figure A.8.

Additionally, we use the judgments for the syntactic sugar in Figure A.9, and typing rules in Figure A.10 as shorthand for the following derivations of judgments involving the syntactic

sugar:

[RAPID]:

$$\frac{\overline{\Gamma\ ;\ -\vdash \pi_i : P_0 \,\&\, P_1 \multimap P_i} \qquad \overline{\Gamma\ ;\ x : P_0 \,\&\, P_1 \vdash x : P_0 \,\&\, P_1}}{\Gamma\ ;\ x : P_0 \,\&\, P_1 \vdash \pi_i \llcorner x : P_i}$$

[APROJ*in*]: By induction on i:

[$i = 0$, $n = m \geq 1$]: Immediate since

$$\begin{aligned} &\Gamma\ ;\ -\vdash \pi_0 : P \,\&\, \&_{m-1} P \multimap P \\ =\ &\Gamma\ ;\ -\vdash \pi_0 : \&_m P \multimap P \\ =\ &\Gamma\ ;\ -\vdash \pi_0^m : \&_m P \multimap P \end{aligned}$$

[$i = 1$, $n = 2$]: Immediate since

$$\begin{aligned} &\Gamma\ ;\ -\vdash \pi_1 : P \,\&\, P \multimap P \\ =\ &\Gamma\ ;\ -\vdash \pi_1 : \&_2 P \multimap P \\ =\ &\Gamma\ ;\ -\vdash \pi_1^2 : \&_2 P \multimap P \end{aligned}$$

[$i = j + 1$, $n = m + 1$, $1 \leq j < m$]: First, trivially

$$\frac{\begin{array}{r}\overline{\Gamma\ ;\ -\vdash \pi_1 : P \,\&\, \&_m P \multimap \&_m P} \\ =\ \Gamma\ ;\ -\vdash \pi_1 : \&_{m+1} P \multimap \&_m P\end{array} \qquad \overline{\Gamma\ ;\ x : \&_{m+1} P \vdash x : \&_{m+1} P}}{\Gamma\ ;\ x : \&_{m+1} P \vdash \pi_1 \llcorner x : \&_m P}$$

And then

$$\frac{\dfrac{\dfrac{\begin{array}{c}\text{the induction hypothesis}\\ \vdots\\ \Gamma\ ;\ -\vdash \pi_j^m : \&_m P \multimap P\end{array} \qquad \Gamma\ ;\ x : \&_{m+1} P \vdash \pi_1 \llcorner x : \&_m P}{\Gamma\ ;\ x : \&_{m+1} P \vdash \pi_j^m \llcorner (\pi_1 \llcorner x) : P}}{\Gamma\ ;\ -\vdash \delta x.\, \pi_j^m \llcorner (\pi_1 \llcorner x) : \&_{m+1} P \multimap P}}{=\ \Gamma\ ;\ -\vdash \pi_{j+1}^{m+1} : \&_{m+1} P \multimap P}$$

[LOOKUP]:

$$\frac{\Gamma\ ;\ \Delta \vdash S : \mathbf{N} \to A \quad \Gamma\ ;\ -\vdash N : \mathbf{N}}{\Gamma\ ;\ \Delta \vdash S\,N : A}$$

[EXTEND]: Let $\Gamma' = \Gamma\,, o : \mathbf{N}$, and

$$\frac{\dfrac{\overline{\Gamma'\ ;\ -\vdash o : \mathbf{N}} \quad \Gamma\ ;\ -\vdash N : \mathbf{N}}{\Gamma'\ ;\ -\vdash o \stackrel{n}{=} N : \mathbf{N}} \qquad \dfrac{\Gamma\ ;\ \Delta \vdash S : \mathbf{N} \to A \quad \overline{\Gamma'\ ;\ -\vdash o : \mathbf{N}}}{\Gamma'\ ;\ \Delta \vdash S[o] : A} \qquad \Gamma\ ;\ \Delta \vdash M : A}{\dfrac{\Gamma'\ ;\ \Delta \vdash o \stackrel{n}{=} N \to S[o] \,[]\, M : A}{\Gamma\ ;\ \Delta \vdash \lambda o.\, o \stackrel{n}{=} N \to S[o] \,[]\, M : \mathbf{N} \to A}}$$

A.3.1 Recursive types formalities

The inclusion of recursive types, and hence type identifiers, makes writing syntactically correct but meaningless types possible, so some extra machinery is required to ensure the types we use are meaningful. The remainder of this section presents the details, following [AF96], of defining the obvious equality on types and giving a formal version of [REC].

A *type context* Σ is a finite set of type identifiers. We use the notation $\Sigma \mathbin{,} \Sigma'$ for the union of Σ and Σ' if Σ and Σ' are disjoint, otherwise $\Sigma \mathbin{,} \Sigma'$ is undefined. The judgment $\Sigma \vdash_{\overline{p}} P$ states that P is a well-formed *pointed type* in type context Σ while the judgment $\Sigma \vdash A$ states that A is a well-formed *type* in type context Σ. The rules defining these judgments are given in Figure A.11, but effectively, a type is well-formed in a type context if its free identifiers are contained in the context and there are no identifier name clashes.

We use the "equality approach" to recursive types to keep types from polluting terms at the expense of defining equality on types. The judgment $\Sigma \vdash A = B$, defined in Figure A.12, states that types A and B are equal in type context Σ. The only rule of interest is the last which indicates that folding or unfolding a recursive type once (and hence finitely-many times) preserves type equality.

The judgment $\Sigma \fatsemi \Gamma \mathbin{;} \Delta \vdash M : A$ states that M is a well-typed term of type A in type context Σ and term context $\Gamma \mathbin{;} \Delta$. In $\Sigma \fatsemi \Gamma \mathbin{;} \Delta \vdash M : A$, the only role Σ plays is to ensure the well-formedness of the types involved, so we usually elide it and use the abbreviations in Figure A.13. Using these, [REC] and the other typing rules take the necessary formalities into account.

Figure A.7 Target Typing Rules

$$[\text{NLIT}]\ \frac{}{\Gamma\ ;\ -\ \vdash n : \mathbf{N}} \qquad [\text{NEQ}]\ \frac{\Gamma\ ;\ \Delta \vdash M : \mathbf{N} \quad \Gamma\ ;\ \Delta' \vdash N : \mathbf{N}}{\Gamma\ ;\ \Delta\ ,\ \Delta' \vdash M \stackrel{n}{=} N : \mathbf{N}}$$

$$[\text{NCOND}]\ \frac{\Gamma\ ;\ \Delta \vdash M : \mathbf{N} \quad \Gamma\ ;\ \Delta' \vdash N : A \quad \Gamma\ ;\ \Delta' \vdash O : A}{\Gamma\ ;\ \Delta\ ,\ \Delta' \vdash M \to N \,[]\, O : A}$$

$$[\text{ID}]\ \frac{}{\Gamma\ ,\ x : A\ ;\ -\ \vdash x : A} \qquad [\text{RID}]\ \frac{}{\Gamma\ ;\ x : P \vdash x : P}$$

$$[\text{ABS}]\ \frac{\Gamma\ ,\ x : A\ ;\ \Delta \vdash M : B}{\Gamma\ ;\ \Delta \vdash \lambda x.\, M : A \to B} \qquad [\text{APP}]\ \frac{\Gamma\ ;\ \Delta \vdash M : A \to B \quad \Gamma\ ;\ - \vdash N : A}{\Gamma\ ;\ \Delta \vdash M\, N : B}$$

$$[\text{RABS}]\ \frac{\Gamma\ ;\ \Delta\ ,\ x : P \vdash M : Q}{\Gamma\ ;\ \Delta \vdash \delta x.\, M : P \multimap Q} \qquad [\text{RAPP}]\ \frac{\Gamma\ ;\ \Delta \vdash M : P \multimap Q \quad \Gamma\ ;\ \Delta' \vdash N : P}{\Gamma\ ;\ \Delta\ ,\ \Delta' \vdash M \_ N : Q}$$

$$[\text{APAIR}]\ \frac{\Gamma\ ;\ \Delta \vdash M : P \quad \Gamma\ ;\ \Delta \vdash N : Q}{\Gamma\ ;\ \Delta \vdash \langle M, N \rangle : P \,\&\, Q} \qquad [\text{APROJ}i]\ \frac{}{\Gamma\ ;\ - \vdash \pi_i : P_0 \,\&\, P_1 \multimap P_i}$$

$$[\text{MPAIR}]\ \frac{\Gamma\ ;\ \Delta \vdash M : P \quad \Gamma\ ;\ \Delta' \vdash N : Q}{\Gamma\ ;\ \Delta\ ,\ \Delta' \vdash (M, N) : P \otimes Q} \qquad [\text{MPATM}]\ \frac{\Gamma\ ;\ \Delta\ ,\ x_0 : P_0\ ,\ x_1 : P_1 \vdash M : Q}{\Gamma\ ;\ \Delta \vdash \delta(x_0, x_1).\, M : P_0 \otimes P_1 \multimap Q}$$

$$[\text{HALT}]\ \frac{}{\Gamma\ ;\ - \vdash \mathsf{halt} : \mathbf{R}} \qquad [\text{UNIT}]\ \frac{}{\Gamma\ ;\ - \vdash \langle\rangle : \mathbf{1}}$$

$$[\text{ABSUNIT}]\ \frac{\Gamma\ ;\ \Delta \vdash M : A}{\Gamma\ ;\ \Delta \vdash \lambda\langle\rangle.\, M : \mathbf{1} \to A} \qquad [\text{RABSUNIT}]\ \frac{\Gamma\ ;\ \Delta \vdash M : P}{\Gamma\ ;\ \Delta \vdash \delta\langle\rangle.\, M : \mathbf{1} \multimap P}$$

$$[\text{REC}]\ \frac{\Gamma\ ;\ \Delta \vdash M : B}{\Gamma\ ;\ \Delta \vdash M : A}\ B = A$$

And for the affine variant:

$$[\text{RWEAK}]\ \frac{\Gamma\ ;\ \Delta \vdash M : B}{\Gamma\ ;\ \Delta\ ,\ x : A \vdash M : B}$$

Figure A.8 Admissible Target Typing Rules

$$[\text{CONT}]\ \frac{\Gamma\ ,\ x : A\ ,\ y : A\ ;\ \Delta \vdash M : B}{\Gamma\ ,\ x : A\ ;\ \Delta \vdash M[y \mapsto x] : B} \qquad [\text{WEAK}]\ \frac{\Gamma\ ;\ \Delta \vdash M : B}{\Gamma\ ,\ x : A\ ;\ \Delta \vdash M : B}$$

$$[\text{EXCH}]\ \frac{\Gamma\ ,\ y : B\ ,\ x : A\ ,\ \Gamma'\ ;\ \Delta \vdash M : C}{\Gamma\ ,\ x : A\ ,\ y : B\ ,\ \Gamma'\ ;\ \Delta \vdash M : C} \qquad [\text{REXCH}]\ \frac{\Gamma\ ;\ \Delta\ ,\ y : B\ ,\ x : A\ ,\ \Delta' \vdash M : C}{\Gamma\ ;\ \Delta\ ,\ x : A\ ,\ y : B\ ,\ \Delta' \vdash M : C}$$

Figure A.9 Target Judgments for Syntactic Sugar

$$\begin{aligned}&\Gamma\ ;\ \Delta\ ,\ \langle x_0, x_1\rangle : P_0 \ \&\ P_1 \vdash M[x \mapsto \langle x_0, x_1\rangle] : P \\ &\quad \stackrel{\text{def}}{=} \Gamma\ ;\ \Delta\ ,\ x : P_0 \ \&\ P_1 \vdash M[x_0, x_1 \mapsto \pi_0\_x, \pi_1\_x] : P\end{aligned}$$

Figure A.10 Target Typing Rules for Syntactic Sugar

$$[\textsc{RAPID}]\ \frac{}{\Gamma\ ;\ \langle x_0, x_1\rangle : P_0 \ \&\ P_1 \vdash x_i : P_i}$$

$$[\textsc{APROJ}in]\ \frac{}{\Gamma\ ;\ - \vdash \pi_i^n : \&_n P \multimap P}\ 0 \le i < n$$

$$[\textsc{LOOKUP}]\ \frac{\Gamma\ ;\ \Delta \vdash S : \mathbf{N} \to A \quad \Gamma\ ;\ - \vdash N : \mathbf{N}}{\Gamma\ ;\ \Delta \vdash S[N] : A}$$

$$[\textsc{EXTEND}]\ \frac{\Gamma\ ;\ \Delta \vdash S : \mathbf{N} \to A \quad \Gamma\ ;\ - \vdash N : \mathbf{N} \quad \Gamma\ ;\ \Delta \vdash M : A}{\Gamma\ ;\ \Delta \vdash [S \mid N{:}M] : \mathbf{N} \to A}$$

Figure A.11 Target Type Well-Formedness Rules

$$\frac{\Sigma \vdash A \quad \Sigma \Vdash_p P}{\Sigma \Vdash_p A \to P} \qquad \frac{\Sigma \Vdash_p P \quad \Sigma \Vdash_p Q}{\Sigma \Vdash_p P \star Q}\ \star \in \{\multimap, \&, \otimes\} \qquad \frac{}{\Sigma \Vdash_p P}\ P \in \{\mathbf{R}, \mathbf{1}\} \qquad \frac{}{\Sigma\ ,\ X \Vdash_p X} \qquad \frac{\Sigma\ ,\ X \Vdash_p P}{\Sigma \Vdash_p \mu X.\, P}$$

$$\frac{}{\Sigma \vdash \mathbf{N}} \qquad \frac{\Sigma \vdash A \quad \Sigma \vdash B}{\Sigma \vdash A \to B} \qquad \frac{\Sigma \Vdash_p P}{\Sigma \vdash P}$$

Figure A.12 Target Type Equality Rules

$$\frac{\Sigma \vdash B = A}{\Sigma \vdash A = B} \qquad \frac{\Sigma \vdash A = C \quad \Sigma \vdash C = B}{\Sigma \vdash A = B}$$

$$\frac{}{\Sigma \vdash A = A}\ A \in \{\mathbf{N}, \mathbf{R}, \mathbf{1}\} \qquad \frac{\Sigma \vdash A = A' \quad \Sigma \vdash B = B'}{\Sigma \vdash A \star B = A' \star B'}\ \star \in \{\to, \multimap, \&, \otimes\}$$

$$\frac{\Sigma \vdash P = Q}{\Sigma \vdash \mu X.\, P = \mu X.\, Q} \qquad \frac{}{\Sigma\ ,\ X \vdash X = X} \qquad \frac{\Sigma\ ,\ X \Vdash_p P}{\Sigma \vdash P[X \mapsto \mu X.\, P] = \mu X.\, P}$$

Figure A.13 Target Type Context Abbreviations

When clear from context

$$\Sigma \fatsemi \Gamma \,;\, \Delta \vdash M : A \qquad \text{is abbreviated} \qquad \Gamma \,;\, \Delta \vdash M : A$$

and

$$\frac{\Sigma \fatsemi \Gamma_1 \,;\, \Delta_1 \vdash M_1 : A_1 \;\cdots\; \Sigma \fatsemi \Gamma_n \,;\, \Delta_n \vdash M_n : A_n \quad \Sigma \vdash A_n = A_0}{\Sigma \fatsemi \Gamma_0 \,;\, \Delta_0 \vdash M_0 : A_0}$$

is abbreviated

$$\frac{\Gamma_1 \,;\, \Delta_1 \vdash M_1 : A_1 \;\cdots\; \Gamma_n \,;\, \Delta_n \vdash M_n : A_n}{\Gamma_0 \,;\, \Delta_0 \vdash M_0 : A_0}\, A_n = A_0$$

A.4 Standard Predomain Semantics

This is not an especially accurate model of the language, because the interpretation of $(\cdot) \multimap (\cdot)$ validates Contraction and Weakening and because the intuitive "abstractness" of the result type is not accounted for. But the model is certainly adequate for any reasonable operational semantics, and so serves as a useful reference point.

Type contexts are interpreted by type environments, which map type identifiers to domains (chain-complete partial orders with least element), as shown in Figure A.14.

For the interpretation of types and terms, the metalanguage notation for (type and term) environments in Figure A.15 will be convenient.

A type judgment is interpreted as a map from a type environment to a predomain (chain-complete partial order), and a pointed type judgment is interpreted as a map from a type environment to a domain, as shown in Figure A.16. We write $\mathbb{N}$ for the discretely ordered predomain of the natural numbers, $(\cdot) \to (\cdot)$ for continuous function space, and $(\cdot) \times (\cdot)$ for Cartesian product. The existence of least solutions of the domain isomorphisms is guaranteed by the inverse limit construction [Sco70].

Contexts are interpreted by maps from type environments to environments: iterated products over the subjects of the context, which map identifiers to elements of the appropriate predomain, as shown in Figure A.17.

Term judgments are interpreted as maps from environments appropriate to the typing context to the predomain appropriate to the type of the term as shown in Figure A.18.

Figure A.14 Semantics of Type Contexts

$$[\![\Sigma]\!] \in \prod_{X \in \mathsf{sub}(\Sigma)} \mathcal{D}om$$

Figure A.15 Metalanguage Syntactic Sugar for Environments

$$
\begin{array}{rll}
[\,] \stackrel{\text{def}}{=} \lambda i.\,\text{undefined} & & \text{empty environment} \\
e[x] \stackrel{\text{def}}{=} e\,x & & \text{environment lookup} \\
[e \mid x\colon d] \stackrel{\text{def}}{=} \lambda i.\begin{cases} d & \text{if } i = x \\ e[i] & \text{otherwise} \end{cases} & & \text{environment extension} \\
e|_{\Xi} \stackrel{\text{def}}{=} \lambda i.\begin{cases} e[i] & \text{if } i \in \mathsf{sub}(\Xi) \\ \text{undefined} & \text{otherwise} \end{cases} & & \text{environment restriction}
\end{array}
$$

Figure A.16 Semantics of Target Types

$$
\begin{array}{rcl}
[\![\Sigma \vdash A]\!] \in [\![\Sigma]\!] \to \mathcal{P}\mathit{redom} & & \\
[\![\Sigma \vdash \mathbf{N}]\!]\,E & \stackrel{\text{def}}{=} & \mathbb{N} \\
[\![\Sigma \vdash A \to B]\!]\,E & \stackrel{\text{def}}{=} & [\![\Sigma \vdash A]\!]\,E \to [\![\Sigma \vdash B]\!]\,E \\
[\![\Sigma \vdash P]\!]\,E & \stackrel{\text{def}}{=} & [\![\Sigma \vdash_{\overline{p}} P]\!]\,E \\
 & & \\
[\![\Sigma \vdash_{\overline{p}} P]\!] \in [\![\Sigma]\!] \to \mathcal{D}\mathit{om} & & \\
[\![\Sigma \vdash_{\overline{p}} A \to P]\!]\,E & \stackrel{\text{def}}{=} & [\![\Sigma \vdash A]\!]\,E \to [\![\Sigma \vdash_{\overline{p}} P]\!]\,E \\
[\![\Sigma \vdash_{\overline{p}} P \multimap Q]\!]\,E & \stackrel{\text{def}}{=} & [\![\Sigma \vdash_{\overline{p}} P]\!]\,E \to [\![\Sigma \vdash_{\overline{p}} Q]\!]\,E \\
[\![\Sigma \vdash_{\overline{p}} P \,\&\, Q]\!]\,E & \stackrel{\text{def}}{=} & [\![\Sigma \vdash_{\overline{p}} P]\!]\,E \times [\![\Sigma \vdash_{\overline{p}} Q]\!]\,E \\
[\![\Sigma \vdash_{\overline{p}} P \otimes Q]\!]\,E & \stackrel{\text{def}}{=} & [\![\Sigma \vdash_{\overline{p}} P]\!]\,E \times [\![\Sigma \vdash_{\overline{p}} Q]\!]\,E \\
[\![\Sigma \vdash_{\overline{p}} \mathbf{R}]\!]\,E & \stackrel{\text{def}}{=} & \{\top\}_{\perp} \\
[\![\Sigma \vdash_{\overline{p}} \mathbf{1}]\!]\,E & \stackrel{\text{def}}{=} & \{\top\}_{\perp} \\
[\![\Sigma\,, X \vdash_{\overline{p}} X]\!]\,E & \stackrel{\text{def}}{=} & E[X] \\
[\![\Sigma \vdash_{\overline{p}} \mu X.\,P]\!]\,E & \stackrel{\text{def}}{=} & \text{the least solution of } D \cong [\![\Sigma\,, X \vdash_{\overline{p}} P]\!]\,[E \mid X\colon D]
\end{array}
$$

Figure A.17 Semantics of Target Contexts

$$
[\![\Sigma \fatsemi \Gamma\,;\,\Delta]\!]\,E \in \prod_{x \in \mathsf{sub}(\Gamma;\Delta)} [\![\Sigma \vdash A]\!]\,E \quad \text{where } E \in [\![\Sigma]\!] \text{ and } x : A \in \Gamma \cup \Delta
$$

Figure A.18 Semantics of Target Terms

$$[\![\Sigma \fatsemi \Gamma \;;\; \Delta \vdash M : A]\!]\, E \in [\![\Sigma \fatsemi \Gamma \;;\; \Delta]\!]\, E \to [\![\Sigma \vdash A]\!]\, E \quad \text{where } E \in [\![\Sigma]\!]$$

$$[\![\Gamma \;;\; - \vdash n : \mathbf{N}]\!]\, E\, e \stackrel{\text{def}}{=} n$$

$$[\![\Gamma \;;\; \Delta \,,\, \Delta' \vdash M \stackrel{n}{=} N : \mathbf{N}]\!]\, E\, e \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if} \quad [\![\Gamma \;;\; \Delta \vdash M : \mathbf{N}]\!]\, E\, e|_{\Gamma;\Delta} \\ & \quad \neq [\![\Gamma \;;\; \Delta' \vdash N : \mathbf{N}]\!]\, E\, e|_{\Gamma;\Delta'} \\ 1 & \text{otherwise} \end{cases}$$

$$[\![\Gamma \;;\; \Delta \,,\, \Delta' \vdash M \to N \,[]\, O : A]\!]\, E\, e \stackrel{\text{def}}{=} \begin{cases} [\![\Gamma \;;\; \Delta' \vdash N : A]\!]\, E\, e|_{\Gamma;\Delta'} \\ \quad \text{if } [\![\Gamma \;;\; \Delta \vdash M : \mathbf{N}]\!]\, E\, e|_{\Gamma;\Delta} = 0 \\ [\![\Gamma \;;\; \Delta' \vdash O : A]\!]\, E\, e|_{\Gamma;\Delta'} \quad \text{otherwise} \end{cases}$$

$$[\![\Gamma \,,\, x : A \;;\; - \vdash x : A]\!]\, E\, e \stackrel{\text{def}}{=} e[x]$$

$$[\![\Gamma \;;\; x : P \vdash x : P]\!]\, E\, e \stackrel{\text{def}}{=} e[x]$$

$$[\![\Sigma \fatsemi \Gamma \;;\; \Delta \vdash \lambda x.\, M : A \to B]\!]\, E\, e \stackrel{\text{def}}{=} \lambda d \in [\![\Sigma \vdash A]\!]\, E.\; [\![\Gamma \,,\, x : A \;;\; \Delta \vdash M : B]\!]\, E\, [e \mid x\colon d]$$

$$[\![\Gamma \;;\; \Delta \vdash M\, N : B]\!]\, E\, e \stackrel{\text{def}}{=} [\![\Gamma \;;\; \Delta \vdash M : A \to B]\!]\, E\, e\; ([\![\Gamma \;;\; - \vdash N : A]\!]\, E\, e|_{\Gamma})$$

$$[\![\Sigma \fatsemi \Gamma \;;\; \Delta \vdash \delta x.\, M : P \multimap Q]\!]\, E\, e \stackrel{\text{def}}{=} \lambda d \in [\![\Sigma \vdash_{\overline{p}} P]\!]\, E.\; [\![\Gamma \;;\; \Delta \,,\, x : P \vdash M : Q]\!]\, E\, [e \mid x\colon d]$$

$$[\![\Gamma \;;\; \Delta \,,\, \Delta' \vdash M \_ N : Q]\!]\, E\, e \stackrel{\text{def}}{=} [\![\Gamma \;;\; \Delta \vdash M : P \multimap Q]\!]\, E\, e|_{\Gamma;\Delta}\; ([\![\Gamma \;;\; \Delta' \vdash N : P]\!]\, E\, e|_{\Gamma;\Delta'})$$

$$[\![\Gamma \;;\; \Delta \vdash \langle M, N \rangle : P \,\&\, Q]\!]\, E\, e \stackrel{\text{def}}{=} ([\![\Gamma \;;\; \Delta \vdash M : P]\!]\, E\, e, [\![\Gamma \;;\; \Delta \vdash N : Q]\!]\, E\, e)$$

$$[\![\Gamma \;;\; - \vdash \pi_i : P_0 \,\&\, P_1 \multimap P_i]\!]\, E\, e \stackrel{\text{def}}{=} \pi_i$$

$$[\![\Gamma \;;\; \Delta \,,\, \Delta' \vdash (M, N) : P \otimes Q]\!]\, E\, e \stackrel{\text{def}}{=} ([\![\Gamma \;;\; \Delta \vdash M : P]\!]\, E\, e|_{\Gamma;\Delta}\; , [\![\Gamma \;;\; \Delta' \vdash N : Q]\!]\, E\, e|_{\Gamma;\Delta'})$$

$$[\![\Sigma \fatsemi \Gamma \;;\; \Delta \vdash \delta(x_0, x_1).\, M : P_0 \otimes P_1 \multimap Q]\!]\, E\, e \stackrel{\text{def}}{=} \lambda d \in [\![\Sigma \vdash_{\overline{p}} P_0 \otimes P_1]\!]\, E.\; [\![\Gamma \;;\; \Delta \,,\, x_0 : P_0 \,,\, x_1 : P_1 \vdash M : Q]\!]\, E\, [[e \mid x_0\colon \pi_0\, d] \mid x_1\colon \pi_1\, d]$$

$$[\![- \;;\; - \vdash \mathsf{halt} : \mathbf{R}]\!]\, E\, e \stackrel{\text{def}}{=} \top$$

$$[\![- \;;\; - \vdash \langle\rangle : \mathbf{1}]\!]\, E\, e \stackrel{\text{def}}{=} \top$$

$$[\![\Sigma \fatsemi \Gamma \;;\; \Delta \vdash \lambda\langle\rangle.\, M : \mathbf{1} \to A]\!]\, E\, e \stackrel{\text{def}}{=} \lambda d \in [\![\Sigma \vdash \mathbf{1}]\!]\, E.\, [\![\Gamma \;;\; \Delta \vdash M : A]\!]\, E\, e$$

$$[\![\Sigma \fatsemi \Gamma \;;\; \Delta \vdash \delta\langle\rangle.\, M : \mathbf{1} \multimap P]\!]\, E\, e \stackrel{\text{def}}{=} \lambda d \in [\![\Sigma \vdash_{\overline{p}} \mathbf{1}]\!]\, E.\, [\![\Gamma \;;\; \Delta \vdash M : P]\!]\, E\, e$$

A.5 Properties

Lemma 33 (Validity of [CONT] **and** [WEAK]**)** *1. If* $\Sigma \fatsemi \Gamma , x : A , y : A \fatsemi \Delta \vdash M : B$*, then for* $E \in [\![\Sigma]\!]$ *and* $e \in [\![\Sigma \fatsemi \Gamma , x : A , y : A \fatsemi \Delta]\!] E$

$$[\![\Sigma \fatsemi \Gamma , x : A , y : A \fatsemi \Delta \vdash M : B]\!] E\, e = [\![\Sigma \fatsemi \Gamma , x : A \fatsemi \Delta \vdash M[y \mapsto x] : B]\!] E\, e|_{\Gamma , x:A}$$

2. If $\Sigma \fatsemi \Gamma \fatsemi \Delta \vdash M : B$*, then for* $E \in [\![\Sigma]\!]$*,* $e \in [\![\Sigma \fatsemi \Gamma \fatsemi \Delta]\!] E$*, and* $d \in [\![\Sigma \vdash A]\!]$

$$[\![\Sigma \fatsemi \Gamma \fatsemi \Delta \vdash M : B]\!] E\, e = [\![\Sigma \fatsemi \Gamma , x : A \fatsemi \Delta \vdash M : B]\!] E\, [e \mid x{:}\, d]$$

Lemma 34 (Validity of $\overset{\beta\eta}{=}$**)** *For any terms* M *and* N *such that* $M \overset{\beta\eta}{=} N$*,* $\Gamma \fatsemi \Delta \vdash M : A$*, and* $\Gamma \fatsemi \Delta \vdash N : A$

$$[\![\Gamma \fatsemi \Delta \vdash M : A]\!] = [\![\Gamma \fatsemi \Delta \vdash N : A]\!]$$

Lemma 35 (Υ **computes least fixed-points)** *For any* M *such that* $\Sigma \fatsemi \Gamma \fatsemi - \vdash M : P \to P$*,* $E \in [\![\Sigma]\!]$*, and* $e \in [\![\Sigma \vdash_{\overline{p}} P]\!] E$

$$[\![\Sigma \fatsemi \Gamma \fatsemi - \vdash \Upsilon M : P]\!] E\, e = \bigsqcup \{ ([\![\Sigma \fatsemi \Gamma \fatsemi - \vdash M : P \to P]\!] E\, e)^n \perp \mid n \geq 0 \}$$

Index

Bibliography

[AF96] Martín Abadi and Marcelo P. Fiore. Syntactic considerations on recursive types. In *Proceedings of 11th Annual IEEE Symposium on Logic in Computer Science, LICS'96, New Brunswick, NJ, USA, 27–30 July 1996*, pages 242–252, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, Washington - Brussels - Tokyo, June 2001. IEEE.

[BBD04a] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations. In Thielecke [Thi04].

[BBD04b] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. Technical Report BRICS RS-04-29, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2004. A preliminary version appeared as [BBD04a].

[BK01] Nick Benton and Andrew Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11(4):395–410, July 2001.

[BL96] Gérard Boudol and Cosimo Laneve. The discriminating power of multiplicities in the λ-calculus. *Information and Computation*, 126(1):83–102, 10 April 1996.

[BORT00] Josh Berdine, Peter W. O'Hearn, Uday S. Reddy, and Hayo Thielecke. Linearly used continuations. In Amr Sabry, editor, *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)*, pages 47–54. Technical Report No. 545, Computer Science Department, Indiana University, December 2000.

[BORT02] Josh Berdine, Peter O'Hearn, Uday Reddy, and Hayo Thielecke. Linear continuation-passing. *Higher-Order and Symbolic Computation*, 15(2/3):181–208, September 2002.

[BOT00] Josh Berdine, Peter O'Hearn, and Hayo Thielecke. Linear use of continuations. Presented at *The Sixteenth Workshop on the Mathematical Foundations of Programming Semantics (MFPS XVI)*, April 2000.

[BOT02] Josh Berdine, Peter O'Hearn, and Hayo Thielecke. Extracting the range of CPS from affine typing: Extended abstract. Presented at *FLoC'02 Workshop on Linear Logic (LL 2002)*, July 2002.

[BP97] Andrew Barber and Gordon Plotkin. Dual intuitionistic linear logic. Also University of Edinburgh Laboratory for Foundations of Computer Science Technical Report ECS-LFCS-96-347, October 1997.

[BWD96] Carl Bruggeman, Oscar Waddel, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 99–107, 1996.

[CHO99] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, April 1999.

[Dan88] Olivier Danvy. On some functional aspects of control. In Th. Johnsson, S. Peyton Jones, and K. Karlsson, editors, *Programming Methodology Group*, pages 445–449, University of Göteborg and Chalmers University of Technology, September 1988. Report 53.

[Dan89] Olivier Danvy. On listing list prefixes. *LISP Pointers*, 2(3-4):42–46, January 1989.

[Dan92] Olivier Danvy. Back to direct style. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 130–150. Springer-Verlag, New York, NY, 1992.

[Dan94] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[Dan00] Olivier Danvy. Formalizing implementation strategies for first-class continuations. *Lecture Notes in Computer Science*, 1782:88–103, 2000.

[Dan04] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Thielecke [Thi04], pages 13–23.

[DDP00] Olivier Danvy, Belmina Dzafic, and Frank Pfenning. On proving syntactic properties of CPS programs. In Andrew Gordon and Andrew Pitts, editors, *Proceedings of HOOTS99, the Third International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*, pages 19–31. Elsevier, 2000.

[DF89] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, Denmark, 1989.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 Conference of LISP and Functional Programming*, pages 151–160. ACM Press, 1990.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[DHM91] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 163–173, Orlando, FL, USA, January 1991. ACM Press.

[DL92] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In *Conference on Lisp and Functional programming*, pages 299–310, San Francisco, USA, June 1992.

[DN01] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174. ACM Press, September 2001.

[DP95] Olivier Danvy and Frank Pfenning. The occurrence of continuation parameters in CPS terms. Technical Report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, February 1995.

[Fel87] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987. The author now prefers the revised theory presented in [FH92].

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *Conference Record of 15th Annual ACM Symposium on Principles of Programming Languages, POPL'88, San Diego, CA, USA, Jan. 1988*, pages 180–190, New York, 1988. ACM Press.

[Fel91] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1–3):35–75, December 1991.

[FFDM87] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Indiana University Computer Science Department, February 1987.

[FH85] Daniel P. Friedman and Christopher T. Haynes. Constraining control. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254. ACM, ACM, January 1985.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[Fil92] Andrzej Filinski. Linear continuations. In Ravi Sethi, editor, *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 27–38, Albuquerque, NM, January 1992. ACM Press.

[Fis72] Michael J. Fischer. Lambda calculus schemata. In *Proceedings of an ACM Conference on Proving Assertions about Programs*, pages 104–109, New York, January 1972. ACM. SIGPLAN Notices, 7(1) and SIGACT News, 14. Preliminary version of [Fis93].

[Fis93] Michael J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6(3/4):259–287, November 1993. Final version of [Fis72].

[FS02] Daniel P. Friedman and Amr Sabry. CPS in little pieces: Composing partial continuations. *Journal of Functional Programming*, 12(6):617–622, 2002. Theoretical Pearl.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6) of *SIGPLAN Notices*, pages 237–247, New York, 1993. ACM Press.

[FW84] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 348–355. ACM, ACM, August 1984.

[FWFD88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Proceedings of the 1988 Conference of LISP and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press, McGraw-Hill Book Company, 1st edition, 1992.

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press, 2nd edition, 2001.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[Has02] Masahito Hasegawa. Linearly used effects: Monadic and CPS transformations into the linear lambda calculus. In *Proceedings 6th International Symposium on Functional and Logic Programming (FLOPS2002)*, volume 2441 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2002.

[HDA94] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson III. Subcontinuations. *LISP and Symbolic Computation*, 7(1):83–110, January 1994.

[HL93] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 206–219, New York, NY, January 1993. ACM.

[Hof00] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, Winter 2000.

[HP79] Matthew C. B. Hennessy and Gordon D. Plotkin. Full abstraction for a simple parallel programming language. In Jiří Bečvář, editor, *Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120, Berlin, September 1979. Springer-Verlag.

[JD88] Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 158–168, San Diego, California, January 1988. ACM.

[Joh87] Gregory F. Johnson. GL – A denotational testbed with continuations and partial continuations as first-class objects. In *Proceedings SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 165–176. ACM, ACM, June 1987.

[KKR+86] David A. Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. ORBIT: An optimizing compiler for scheme. In *Proceedings of SIGPLAN '86 Symposium on Compiler Construction*, volume 21(7), pages 219–233, July 1986.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[KYD98] Jung-taek Kim, Kwangkeun Yi, and Olivier Danvy. Assessing the overhead of ML exceptions by selective CPS transformation. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, pages 103–114, 1998.

[Lai03] Jim Laird. A game semantics of linearly used continuations. extended abstract. In *Proceedings of Foundations of Software Science and Computation Structures, FoSSaCS*, 2003.

[Lan65] Peter J. Landin. A generalization of jumps and labels. Report, UNIVAC Systems Programming Research, August 1965. Reprinted as [Lan98].

[Lan98] Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, December 1998. Reprint of [Lan65], introduced by [Thi98].

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[Mor70] Lockwood F. Morris. The next 700 programming language descriptions. Later published as [Mor93], November 1970.

[Mor93] Lockwood F. Morris. The next 700 formal language descriptions. *LISP and Symbolic Computation*, 6(3/4):249–258, November 1993. Published version of [Mor70].

[MQ94] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, A duumvirate of control operators. In Manuel Hermenegildo and Jaan Penjam, editors, *International Conference on Programming Language Implementation and Logic Programming (PLILP'94). Proceedings*, volume 844 of *Lecture Notes in Computer Science*, pages 182–197. Springer-Verlag, 1994.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[MW85] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi. In Rohit Parikh, editor, *Proceedings of 3rd Workshop on Logics of Programs, Brooklyn, NY, USA, 17–19 June 1985*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224, Brooklyn, June 1985. Springer-Verlag.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[Nec97] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[OR00] Peter W. O'Hearn and John C. Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the ACM*, 47(1):167–223, January 2000.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[Plo76] Gordon D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, September 1976.

[PP00] Jeff Polakow and Frank Pfenning. Properties of terms in continuation-passing style in an ordered logical framework. In Joëlle Despeyroux, editor, *Workshop on Logical Frameworks and Meta-Languages (LFM 2000)*, June 2000.

[PY01] Jeff Polakow and Kwangkeun Yi. Proving syntactic properties of exceptions in an ordered logical framework. In Herbert Kuchen and Kazunori Ueda, editors, *Functional and Logic Programming: 5th International Symposium, FLOPS 2001*, volume 2024 of *Lecture Notes in Computer Science*, pages 61–77. Springer-Verlag, 2001.

[Rey70] John C. Reynolds. GEDANKEN – a simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, 13(5):308–319, May 1970.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, volume 2, pages 717–740, New York, August 1972. ACM. Reprinted as [Rey98a].

[Rey98a] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998. Reprint of [Rey72], with a foreword.

[Rey98b] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Los Alamitos, California, 2002. IEEE Computer Society.

[RT99] Jon G. Riecke and Hayo Thielecke. Typed exceptions and continuations cannot macro-express each other. In Jǐrí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Proceedings 26th International Colloquium on Automata, Languages and Programming (ICALP)*, number 1644 in Lecture Notes in Computer Science, pages 635–644. Springer-Verlag, 1999.

[Sco70] Dana S. Scott. Outline of a mathematical theory of computation. Technical Monograph PRG-2, Programming Research Group, Oxford University Computing Laboratory, November 1970.

[SF90] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *LISP and Symbolic Computation*, 3(1):67–99, January 1990.

[SF93] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3/4):289–360, 1993.

[Sha04] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 5th workshop on Scheme and functional programming*, pages 99–107. ACM SIGPLAN, 2004.

[SS72] Joseph E. Stoy and Christopher Strachey. OS6—an experimental operating system for a small computer. part 1: General principles and structure. *The Computer Journal*, 15(2):117–124, 1972.

[SS76] Guy Lewis Steele, Jr. and Gerald Jay Sussman. Lambda: The ultimate imperative. AI Lab Memo AIM-353, MIT AI Lab, March 1976.

[SS78] Guy Lewis Steele, Jr. and Gerald Jay Sussman. The art of the interpreter of, the modularity complex (parts zero, one, and two). AI Lab Memo AIM-453, MIT AI Lab, May 1978.

[SS79] Guy Lewis Steele, Jr. and Gerald Jay Sussman. Design of LISP-based processors, or SCHEME: A dielectric LISP, or finite memories considered harmful, or LAMBDA: The ultimate opcode. AI Lab Memo AIM-514, MIT AI Lab, March 1979.

[SS80] Guy Lewis Steele, Jr. and Gerald Jay Sussman. Design of a LISP-based microprocessor. *Communications of the ACM*, 23(11):628–645, 1980.

[SS98] Gerald J. Sussman and Guy L. Steele, Jr. Scheme: An interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998. Reprinted from the AI Memo 349, MIT (1975), with a foreword.

[Ste76] Guy Lewis Steele, Jr. Lambda: The ultimate declarative. AI Lab Memo AIM-379, MIT AI Lab, November 1976.

[Ste77a] Guy Lewis Steele, Jr. Debunking the 'expensive procedure call' myth, or, procedure call implementations considered harmful, or, LAMBDA: The ultimate GOTO. AI Lab Memo AIM-443, MIT AI Lab, October 1977.

[Ste77b] Guy Lewis Steele, Jr. Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or LAMBDA, the ultimate GOTO. In *Proceedings of the 1977 annual conference*, pages 153–162. ACM, 1977.

[Ste78] Guy Lewis Steele, Jr. RABBIT: A compiler for SCHEME. AI Lab Technical Report AITR-474, Massachusetts Institute of Technology, May 1978.

[Ste80] Guy Lewis Steele, Jr. Compiler optimization based on viewing LAMBDA as RENAME + GOTO. AI: An MIT Perspective, 1980.

[SW74] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG–11, Programming Research Group, Oxford University Computing Laboratory, January 1974. Reprinted as [SW00].

[SW00] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, April 2000. Reprint of [SW74], with a foreword.

[Thi97] Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997.

[Thi98] Hayo Thielecke. An introduction to Landin's "A generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998. An introduction to [Lan98].

[Thi99a] Hayo Thielecke. Continuations, functions and jumps. *SIGACT News*, 30(2):33–42, June 1999.

[Thi99b] Hayo Thielecke. Using a continuation twice and its implications for the expressive power of `call/cc`. *Higher-Order and Symbolic Computation*, 12(1):47–74, 1999.

[Thi00] Hayo Thielecke. On exceptions versus continuations in the presence of state. In Gert Smolka, editor, *Programming Languages and Systems: 9th European Symposium on Programming, ESOP 2000. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000. Proceedings.*, volume 1782 of *Lecture Notes in Computer Science*, pages 397–411. Springer-Verlag, 2000.

[Thi02] Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2/3):141–160, September 2002.

[Thi03] Hayo Thielecke. From control effects to typed continuation passing. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 139–149. ACM Press, 2003.

[Thi04] Hayo Thielecke, editor. 'Proceedings of the Fourth ACM SIGPLAN Continuations Workshop (CW'04)'. Technical Report No. CSR-04-1, School of Computer Science, University of Birmingham, 2004.

[TMC+96] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[Wad91] Philip Wadler. Is there a use for linear logic? In *Proceedings of the 1991 ACM SIGPLAN symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 255–273, New Haven, Connecticut, United States, June 1991. ACM Press.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In Ruth E. Davis and John R. Allen, editors, *Conference Record of the 1980 LISP Conference*, pages 19–28, August 1980. Reprinted as [Wan99].

[Wan99] Mitchell Wand. Continuation-based multiprocessing. *Higher-Order and Symbolic Computation*, 12(3):285–299, October 1999. Reprint of [Wan80], with a foreword.

[WM01] David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177+, 2001.

[ZM02] Steve Zdancewic and Andrew C. Myers. Secure information flow and linear continuations. *Higher-Order and Symbolic Computation*, 15(2/3):209–234, September 2002.