RICE UNIVERSITY

Models of Control and Their Implications for Programming Language Design

by

Dorai Sitaram

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Matthias Felleisen
Professor of Computer Science

Robert S. Cartwright
Professor of Computer Science

Klaus Weissenberger
Professor of German and Slavic Studies

Houston, Texas

April, 1994

Models of Control and Their Implications for Programming Language Design

Dorai Sitaram

Abstract

This work uses denotational models to understand and improve the part of a programming language concerned with non-local control operators. These operators let the programmer identify and restore arbitrary control contexts in the program execution path, and thus form a powerful component of many modern programming languages.

We use a variety of denotational models to tackle the issues of (1) describing a control language mathematically, and (2) using the model's apparatus to obtain information useful for designing the language. For this, the full abstraction criterion of testing a model against a language is viewed as a feedback loop that suggests language changes. The results from radically different models, for a variety of control manipulation languages, uniformly emphasize the need for delimiting control actions. In the case of higher-order control, this takes the form of a systematic handling of control objects. To check the pragmatics of the new control techniques, we present an implementation and many examples where these delimiters and handlers provide elegant solutions.

Acknowledgments

I am deeply grateful to:

— Matthias Felleisen, for being a most supportive mentor and friend; for taking me to *call/cc* and beyond; for his unerring good taste; for his patience and counsel at times when I saw no way out; and for his willingness to work with me with enthusiasm and compassion despite my falls and blunders.

— Robert "Corky" Cartwright, for his friendship and support throughout my graduate studies—his academic rigor and unfailing geniality are an inspiration.

— Klaus Weissenberger, for being my teacher in the humanities and for taking time out from his busy schedule to be my third reader.

— Bruce Duba and Rama Kanneganti, for several lively discussions.

— Ivy Jorgensen and Rhonda Guajardo, for their kindness and help in shepherding me through several administrative turns and for looking out for me in Houston, Texas, as I wrote my thesis more than a thousand miles away, in Worcester, Massachusetts.

— My grandparents, Meenakshi (†1992) and R. Vanchinathan (†1994), for their affectionate support for a work whose final form they would not see.

— V. Sivasubramanian, for his unstinting help in launching my graduate studies.

— My parents, brothers, and sister, for their faith in me.

— Margaret Wong, my dearest wife, successful fellow doctoral candidate, overworked teacher and breadwinner, for everything: her attention and care despite the demands of her own work and dissertation; her great good humor in the face of all odds; her serenity; and her love.

Contents

Chapter 1

Introduction

This work uses denotational models to study and enhance programming language constructs that manipulate non-local context information. Non-local control operators occur in such diverse languages as Algol [12], Lisp [34, 49], C [30], C++ [53], Scheme [5, 54], ML [23, 37], EuLisp [38] and Dylan [11]—they enable the programmer to identify and restore enclosing or arbitrary control contexts in the program execution path. We show here that widely differing denotational models for any kind of control language invariably include facilities for delimiting and handling control actions—even though the language itself does not. The analysis thus suggests including these facilities in the language and thereby restructuring the latter's control apparatus. Investigations into the pragmatics of control delimiters and handlers demonstrate that these constructs are both useful and implementable.

1.1 Non-local control operators

The simplest instance of a non-local control operator is one that *exits* or *aborts* a running computation, by terminating the entire program or a subcomputation. Such an aborting or *first-order* control construct allows the user to cut short a running program, procedure, loop or similar block of code, once it is determined that the result sought is either already obtained or is no longer interesting. To do this, the construct transfers control to a dynamically enclosing control point.

In contrast, the more powerful *higher-order* control forms store away arbitrary control states as "continuation" abstractions that can be invoked *anywhere* in the program to revisit the stored control state. The act of seizing the information about the control state of the program and providing it to the user is called control *reification*: Hence, these operators are also known as *control reifiers*.

Both forms of non-local control manipulation are powerful programming tools. First-order control is the foundation for the sophisticated exception- or condition-handling systems in languages such as Common Lisp, ML, C++, EuLisp, and Dylan. Higher-order control operators provide a rich array of control metaphors—e.g., coroutines, task scheduling, stream processing, and non-blind backtracking [24, 25, 26, 27]. In a quantifiable sense, control operators are *expressive* [18]: Neither purely functional languages nor those augmented with an assignment operator can simulate control actions with local syntactic translations. As can be expected, higher-order control operators are even more expressive than their first-order counterparts.

1.2 Prior motivating work

The expressiveness of control operators comes at a cost—they invalidate many of the familiar laws of reasoning that hold for functional languages [14]. As Meyer and Riecke observed, terms that once behaved identically now act very differently, suggesting that control operators are "unreasonable" [35]. In technical terms, the *operational equivalence* relation that identifies interchangeable terms for a functional language is *not* a subset of the same relation for the language extended with a control operator. This is inevitable, for a language enhancement that did not break operational equivalences would not be expressive, and thus cannot be an "enhancement."

On the other hand, although control operators are an enhancement, we have no external evidence that our choice of enhancement is the right one. The advanced control operators in languages such as Scheme and ML are too useful to ignore, but often their very power makes them difficult to manage and to understand. Too often, a heavy price in terms of excessive bookkeeping code and makeshift modularity is paid. For example, it is difficult to keep multiple uses of control idioms from interfering, or to enforce arbitrary patterns of desired interference. Haynes, Friedman and Wand [25, 26, 27] present several sophisticated examples of programming that addresses the problem of "constraining control," but because of its ad hoc nature, and its willing sacrifice of the more powerful raw control operator in favor of one of its constrained uses, more satisfactory techniques are needed.

1.3 Denotational models

Denotational or *mathematical models* [43] provide a mathematically amenable reasoning tool for a programming language. A denotational model offers a *meaning function* that maps the syntactic phrases ("*terms*") of the language to mathematical objects called *meanings* or *denotations*. The mapping is defined *compositionally*: In other words, a term's meaning depends only on the meanings—and no other property—of its subterms. The meanings mapped to are elements in a collection called the *domain*.

A model determines *two* natural equivalence relations on the terms in the language [36, 39]. First, two terms are deemed *denotationally equivalent* if they map to the same meaning or denotation. The second relation focuses on the behavior of the terms *as it appears to a user*, who can only observe the working of whole programs. Thus, two terms are *observationally equivalent* if each can be substituted for the other in any program whatsoever without affecting that program's behavior. (For our purposes, the only program behavior of interest is *whether* the program *halts*, producing some value as an answer.) To the user of the program, observational equivalence is the more relevant of the two relations, since it identifies precisely those terms that cannot be told apart in reality. It is also the more complex relation, since it relies on the interchangeability of terms in arbitrary program contexts. The model's own relation, denotational equivalence, though easily defined, is unavoidably a descriptor of the model rather than the language. The first step toward using the model is to connect the two equivalences in some tractable way. Ideally, the two relations should describe the same equivalence, a property called *full abstraction of the model* [36, 39].

A *sound* model ensures that the assignment of meanings does not result in any *observable* inconsistency. In other words, if the model assigns the same meaning to two terms, then a program context filled with either term should have the same meaning. This follows immediately from compositionality, and thus establishes the theorem that denotationally equivalent terms are also observationally equivalent. The converse typically fails for the general run of languages and models—there are many instances of observationally indistinguishable terms that nevertheless map to different denotations. For example, in his treatment of the typed λ-calculus language PCF [39], Plotkin showed that the conventional basic direct model is *not* fully abstract, by presenting two terms that are observationally, but not denotationally, equivalent. In essence, the program contexts possible in the language syntax are not powerful enough to bring forth the difference between the terms *as a program answer*. On the other hand, the richer mathematical functions available in the model easily distinguish the different denotations. The reason is that the usual models of purely functional languages contain a deterministic parallel operator that is not available in the language. This operator, parallel-**or**, produces the result of one of its two argument computations only if the other computation yields the same result or is guaranteed to loop forever. It is tempting to consider adding this feature to the language, and thus cancel the mismatch between language and model, were it not for the fact that it is an expensive addition and of little or no utility. In the case of non-local control, however, the lessons to be gleaned from denotational models are far more pragmatic.

1.4 Denotational models for control

In this thesis, we study the full abstraction results of two quite different models for a variety of control languages. We first use a functional substrate language called Pure Scheme (PS) that has no type restrictions and can be enhanced with control operators that are first- or higher-order. The purely functional language is an idealized but powerful, nonimperative subset of the real language Scheme: hence the name. The basic direct model that describes PS consists of a reflexive domain constructed using Scott's information systems [43] and the traditional mapping of syntax to semantics for the λ-calculus.

The control models are of two kinds: The first is the traditional continuation semantics propounded by Strachey and Wadsworth [52]. The second is an extension of the usual basic direct model used for the functional subset, and was recently proposed by Felleisen and Cartwright [19]. The continuation model, also called the cps (for continuation-passing-style) model, provides a straightforward description of the language PS enhanced with *call-with-current-continuation* (or *call/cc*), the canonical higher-order control reifier present in Scheme and ML. The model describes control operations by including an additional, "continuation" argument to the model's meaning function. In effect, the continuation semantics encodes a stack machine for the language, with the continuation being a functional representation of the control stack. On the other hand, the extended direct model does not rely on the continuation-passing technique but builds the required information for control operations upon demand.

Like the basic direct model, all the control models also possess the ability to perform deterministic parallel disjunction. However, they also demonstrate *control-specific* capabilities that are not present in the modeled language. For instance, the extended direct model for a functional language enhanced with **abort** also has the ability to *delimit* aborting computations. Arbitrarily delimitable aborts are not just a theoretical necessity to ensure full abstraction—they are widely used in languages with first-order control, e.g., Lisp 1.5's **errorset** and *error* [34], Common Lisp's **catch** and **throw** [49] and ML's **handle** and **raise** [23, 37]. A common embellishment for this setup is the association of tags on both abort and delimiter—e.g., a tagged **throw** can be delimited only a **catch** with an identical tag.

More interestingly, we show that the traditional continuation model for *higher-order* control operators such as Scheme's *call/cc* also has control delimiters. First introduced as an operationally useful mechanism by Felleisen [16], higher-order delimiters, also called *prompts*, have been suggested in various settings. Examples include the **reset** of Danvy and Filinski [8], Hieb, Dybvig and Anderson's *spawn* [28, 29], and Queinnec and Serpette's *splitter* [40]. In all cases, the delimiter endows the programmer with the ability to run a subcomputation as an independent program, *in so far as control actions are concerned*. The uses of the prompt are diverse: it allows cleaner realizations of existing control abstractions and makes possible completely new programming styles [16, 46].

As a radically different model for higher-order control, we use Felleisen and Cartwright's technique for extending the basic direct model to our control sublanguages. Despite these models' different approach to modeling control, they too are found to contain prompts! This, combined with the results for the cps model and the direct model for first-order control, is compelling evidence that control-delimiting goes hand-in-hand with control-manipulation, whatever the flavor of the control structure.

An important design clue suggested by the extended models for higher-order control is that higher-order control delimiters are control-*handling* delimiters, thereby obviating the need for the control reifer to do anything but capture the control object. (In contrast, conventional control reifiers like *call/cc* are forced to perform some ad hoc control handling as well, owing to the absence of real handlers.) A control delimiter handles both an aborted object *and* the continuation object created during the control act. This forms an apt generalization of first-order control, where the prompt handles a single aborted object, and where control acts do not create continuations. Furthermore, control handlers are true higher-order versions of the popular exception-handling mechanisms.

1.5 Publications of results

Components of this thesis and some related results were published separately. In "Control Delimiters and Their Hierarchies" [46], we showed that control delimiters allow cleaner realizations of existing control abstractions and make possible completely new programming styles. In "Reasoning with Continuations II: How to Get Full Abstraction for Models of Control" [47], we presented the full abstraction result for the basic direct model for PCF_V, a *call-by-value* version of the typed λ-calculus. The same work also describes the extended model for PCF_V enhanced with the first-order **abort** construct, and a typed cps model for PCF_V augmented with the higher-order control operator *call/cc*. All three models possess the call-by-value variant of Plotkin's parallel-**or**. In addition, the control models also delimit control actions, be they first- or higher-order. In "Modeling Continuations without Continuations" [48], we presented a preliminary investigation of reflexive models for PS, a *latently* typed language and its control extensions. In particular, we used the Felleisen and Cartwright extension technique to create control models that are very similar to the basic direct model. These typeless models not only describe a more realistic and powerful language a la Scheme than [47], but also provide additional insight into the form of higher-order control delimiting—including control handling, a higher-order generalization of first-order exception handling and condition systems.

In "Handling Control" [45], we readdressed pragmatic issues, and described important classes of use where the higher-order control-handling approach comes into its own.

1.6 Outline

The following chapter describes the prototype bare-bones functional language Pure Scheme or PS, the vehicle for the upcoming discussion. Several control enhancements to PS are introduced in an operational setting. Chapter 3 defines a mathematical model for PS and introduces the notation and proof techniques for full abstraction results, and the feedback loop involved in using the results to extend the language in order to obtain full abstraction. The mathematical background for domain construction is presented in more detail in the Appendix. Chapter 4 introduces the continuation-passing-style model, the model of choice for languages with control. The language studied is PS enhanced with *call/cc*, i.e., the functional and control subset of the real language Scheme. Delimiters arise as a consequence of the quest for full abstraction. Chapter 5 studies extensions of the basic direct model of Chapter 3, the languages studied being both first-order and higher-order control enhancements of PS. In both cases, delimiters are crucial in bridging the gap between model and language. In particular, higher-order control needs a handling delimiter that handles not just aborted objects but also continuation objects.

The next two chapters address the pragmatic benefits of control delimiters: Chapter 6 illustrates how the enhanced control operators open up the control information in a program to a user, giving rise to more transparent versions of control-based code or to new control abstractions altogether. Chapter 7 explains how the new control team may be implemented, in particular, their embedding in a language with assignment and a Scheme-like *call/cc*. The final chapter summarizes contemporary work and conclusions.

Chapter 2

Control operators in languages

We describe the syntax and behavior of a spare but powerful language that serves as the vehicle for our theoretical investigations. First, we introduce a purely *functional* subset of the language, and then present various *control* extensions of it. Finally, for each extended version of the language, we describe an abstract machine for evaluating the programs in the language.

2.1 Pure Scheme

We use *Pure Scheme* or *PS* as the prototypical language to present control manipulation. PS is the purely functional subset of the programming language Scheme [5, 54], and is essentially the dynamically typed call-by-value λ-calculus [3, 39] with integers and some elementary arithmetic procedures. A control extension of PS is plain PS enhanced with one or more control-manipulating constructs.

Figure 2.1 describes the syntax of the language PS. The *terms* of PS are either *values* or *non-values*: The values are either constants, variables, or **lambda**-abstractions. The non-values are procedure applications. Constants are further subdivided into integer numerals and some primitive procedure names. Both primitive procedures and **lambda**-abstractions are *procedure values*.

The numerals stand for the respective integers. If n is metasyntax for a number, then $\ulcorner n \urcorner$ is the corresponding numeral. The primitive procedures are the conditional *ef* and simple arithmetic functions: *add1* and *sub1* for incrementing and decrementing a number respectively; *number?* for testing numberhood. The **lambda**-abstractions

Syntax

$$
\begin{array}{rcll}
M & ::= & V & ;\text{values} \\
  & | & (M\ M) & ;\text{applications} \\
V & ::= & \mathsf{c} & ;\text{constants} \\
  & | & x & ;\text{variables} \\
  & | & (\textbf{lambda}\ (x)\ M) & ;\textbf{lambda}\text{-abstractions} \\
\mathsf{c} & ::= & \ulcorner n \urcorner & ;\text{numerals} \\
  & | & \mathsf{f} & ;\text{basic procedure names} \\
\mathsf{f} & ::= & \mathit{add1} \mid \mathit{sub1} \mid \mathit{number?} \mid \mathit{ef} &
\end{array}
$$

Reduction rules

$$
\begin{array}{rcll}
(\mathit{add1}\ \ulcorner n \urcorner) & \longrightarrow & \ulcorner n+1 \urcorner & \\
(\mathit{sub1}\ \ulcorner n+1 \urcorner) & \longrightarrow & \ulcorner n \urcorner & \\
(\mathit{number?}\ \ulcorner n \urcorner) & \longrightarrow & 1 & \\
(\mathit{number?}\ V) & \longrightarrow & 0 & ;V \text{ is not a numeral} \\
(\mathit{ef}\ 0) & \longrightarrow & (\textbf{lambda}\ (x\ y)\ y) & \\
(\mathit{ef}\ V) & \longrightarrow & (\textbf{lambda}\ (x\ y)\ x) & ;V \text{ is not } 0 \\
((\textbf{lambda}\ (x)\ M)\ V) & \longrightarrow & M[x/V] & ;\beta\text{-value rule}
\end{array}
$$

Figure 2.1 Syntax and reduction rules of plain PS.

build procedures, introducing the procedure's parameter as a *bound* variable in the text of the procedure body. Those variables in a term that are not bound are *free.* Terms with no free variables are deemed *closed*—others are *open.* PS programs are exactly the closed terms.

The notation $M[x/N]$ represents the term M with the term N *substituting* all occurrences of the free variable x in N. Finally, a *context* $C[\ ]$ is a term with a "hole" where a *subterm* should be; $C[M]$ is the term that results from filling the hole in context $C[\ ]$ with the subterm M, possibly binding some or all of the free variables, if any, in M. Later, we shall also use *generalized contexts*, which have zero or more holes, each fillable by a subterm. For any term M, a *program context* is a context $C[\,]$ such that $C[M]$ is a program, i.e., a closed term.

2.2 An abstract machine for PS

Figure 2.1 also describes the behavior, or *operational semantics*, of PS. The set of primitive reduction rules describes an abstract machine [15, 20]. The chief reduction rule, the *β-value* rule, is the application of a **lambda**-procedure to an argument value—this produces the body of the procedure, with the argument substituted for the bound variable. The β-value rule asserts that procedures can only be applied to values; thus, PS is a *call-by-value* language.

The procedural constant *add1* adds one to its numeral argument; *sub1* subtracts one from its. The primitive predicate *number?* checks if its argument value is a number. The conditional *ef* picks its second or third argument value according as its first argument value is *true* or *false*—to this end, it treats zero as false, and every other value as true.

We use a handful of syntactic abbreviations for convenience: The procedure $(\textbf{lambda}\ (x\ y\ \ldots)\ M)$ abbreviates $(\textbf{lambda}\ (x)\ (\textbf{lambda}\ (y)\ \ldots\ M))$. The list $(M\ N\ P\ \ldots)$ is a cascaded application associated to the left, i.e., $(((M\ N)\ P)\ \ldots)$. The familiar conditional *form* **if** is syntactic sugar based on *ef*:

$$(\textbf{if}\ M\ N\ P) \stackrel{\mathrm{df}}{=} ((ef\ M\ (\textbf{lambda}\ (d)\ N)\ (\textbf{lambda}\ (d)\ P))\ 0)$$

where d is a variable free in neither N nor P. Procedures such as $(\textbf{lambda}\ (d)\ N)$ that have a dummy argument parameter are called *thunks*. Invoking them on a dummy value—say 0—“discharges” the thunk, causing the behavior of the thunk body. Other common abbreviations are **begin** and **let**:

$$(\textbf{begin}\ M) \stackrel{\text{df}}{=} M$$
$$(\textbf{begin}\ M\ N\ \ldots) \stackrel{\text{df}}{=} ((\textbf{lambda}\ (d)\ (\textbf{begin}\ N\ \ldots))\ M)$$
$$\text{;where } d \text{ is not free in } N, \ldots$$

$$(\textbf{let}\ ([x\ M]\ \ldots)\ N) \stackrel{\text{df}}{=} ((\textbf{lambda}\ (x\ \ldots)\ N)\ M\ \ldots)$$

A **begin**-expression evaluates its operands in sequence. A **let**-expression introduces local variables for hidden use within a textual "block" of code.

Programs are *evaluated* by finding a *redex*—a subterm that matches the left-hand side of some reduction rule—and reducing it. The search for the appropriate redex is guided by splitting the program term into an *evaluation context* and the redex in its hole. An evaluation context is a special kind of context with the following grammar:

$$E[\,] ::= [\,] \mid (V\ E[\,]) \mid (E[\,]\ M)$$

Thus, the innermost, leftmost redex is picked, so PS applications evaluate left-to-right. Applying the relevant reduction rule on the redex—retaining the context—gives a transformed program. This single evaluation *step* is defined as a function on programs:

$$E[M] \rhd E[M'] \qquad \text{if} \qquad M \longrightarrow M'$$

The stepping process is continued until the program can be reduced no more, i.e., it has been reduced to a value, which is the "answer" or "result" of the program. This whole program evaluation process can be abstracted by the partial function *eval* from programs to values:

$$\mathit{eval}(M) = V, \qquad \text{if} \qquad M \stackrel{*}{\rhd} V$$

Programs for which the evaluation process fails to yield a value are called *non-terminating*, *non-converging* or *diverging*. An example of such a program is:

$$\Omega \stackrel{\mathrm{df}}{=} ((\mathbf{lambda}\ (x)\ (x\ x))\ (\mathbf{lambda}\ (x)\ (x\ x)))$$

Plainly, Ω—or indeed, *any* program with Ω in an evaluation context—steps forever: $\Omega \triangleright \Omega \triangleright \ldots$, ad infinitum.

2.3 Activity

Evaluation contexts provide useful clues about the subterms that contribute to the result of a program. This information is captured succinctly in the form of an *activity lemma.* The formulation and proof of activity used here is a refinement of Plotkin's activity lemma for PCF [39] due to Felleisen [17]. We first introduce two definitions.

Definition 2.1 *(Generalized contexts)* A *generalized context* is like a context except that it can have zero or more holes. A useful notation for an m-hole generalized context is $C[\ ]_1 \ldots [\ ]_m$, where each $[\ ]_i$ represents a hole. Filling this context with M_i in the ith hole gives the term $C[M_1]_1 \ldots [M_m]_m$. Generalized contexts satisfy the grammar:

$$C ::= [\ ] \mid x \mid \mathsf{c} \mid (C\ C) \mid (\mathbf{lambda}\ (x)\ C)$$

Definition 2.2 *(Kind)* Two closed terms M and N are of the *same kind* if they are both applications, both **lambda**-abstractions or both observables. (Note that although we treat variables as values, neither M nor N can be a variable since they are closed.)

We now present a couple of lemmas used in proving the activity lemma.

Lemma 2.1 If f is a primitive procedure, then for all procedure values V, either $(\mathsf{f}\ V)$ is irreducible or reduces to the same value W.

Proof The primitive procedures are *add1*, *sub1*, *number?*, and *ef*. Clearly, the first two are irreducible for a procedure argument; *number?* always yields false; and *ef* always takes the true branch. ∎

Lemma 2.2 If $E[\,] = C_1[C_2[\,]]$ where $E[\,]$ is an evaluation context, then both $C_1[\,]$ and $C_2[\,]$ are evaluation contexts.

Proof The proof is an induction on the structure of $C_1[\,]$.

1. Let $C_1[\ ] = [\ ]$. Clearly, $C_1[\ ]$ is an evaluation context. $C_2[\ ] = E[\ ]$ and is therefore an evaluation context.

2. Let $C_1[\ ] = (N\ C_1'[\ ])$. Then, $E[\ ] = (N\ C_1'[C_2[\ ]])$. From the definition of evaluation contexts, N must be a value, and $C_1'[C_2[\ ]]$ an evaluation context. By induction, $C_1'[\ ]$ and $C_2[\ ]$ are evaluation contexts. Since N is a value, $(N\ C_1'[\ ])$ is an evaluation context.

3. Let $C_1[\ ] = (C_1'[\ ]\ N)$. Then $E[M] = (C_1'[C_2[\ ]]\ N)$. Therefore, $C_1'[C_2[\ ]]$ is an evaluation context, and hence, by induction $C_1'[\ ]$ and $C_2[\ ]$ are too. Thus, $(C_1'[\ ]\ N)$ is an evaluation context too.

4. Let $C_1[\,] = ($**lambda** $(x)\ C_1'[\,])$. Then $E[\,] = ($**lambda** $(x)\ C_1'[C_2[\,]])$, which is impossible.

Thus, for all possible $E[\,] = C_1[C_2[\,]]$, $C_1[\,]$ and $C_2[\,]$ are evaluation contexts. ∎

We are now ready for the activity lemma.

Lemma 2.3 *(Activity)* If $C[\ ]_1 \ldots [\ ]_m$ is an m-hole context, and the program $P = C[M_1]_1 \ldots [M_m]_m$ for closed expressions M_i converges, then

1. for all closed M_i' that are of the same kind as the respective M_i, the program $P' = C[M_1']_1 \dots [M_m']_m$ also converges; or
2. for all closed M_i' that are of the same kind as M_i, there is some i between 1 and m and an evaluation context $E'[\,]$ such that
   (a) M_i is an application and
   $$P' = C[M_1']_1 \dots [M_m']_m \stackrel{*}{\triangleright} E'[M_i']$$
   (b) M_i is a **lambda**-abstraction and
   $$P' = C[M_1']_1 \dots [M_m']_m \stackrel{*}{\triangleright} E'[(M_i'\ U)]$$
   for some value U,
   (c) M_i is an observable and
   $$P' = C[M_1']_1 \dots [M_m']_m \stackrel{*}{\triangleright} E'[(\mathsf{f}\ M_i')]$$
   for some primitive procedure f.

Proof The proof is an induction on the number of steps required for the program P to converge to a value, say V. If P itself is a value, then P' is also a value, and (1) is satisfied. If P is *not* a value, then it can be decomposed into an evaluation context $E[\,]$ and redex $(Q\ R)$ as follows:

$$P = C[M_1]_1 \dots [M_m]_m = E[(Q\ R)]$$

Q is a procedure value and R is a value. There are three broad cases and several subcases:

1. $(Q\ R)$ is contained entirely within one of the M_i.
2. $(Q\ R)$ contains some of the M_i. Subcases are:

(a) Q is one of the M_i.

(b) Q is a **lambda**-abstraction possibly containing some of the M_i.

(c) Q occurs outside all the M_i and is a primitive procedure. Subcases are:

i. R is one of the M_i.

ii. R is a **lambda**-abstraction containing one or more of the M_i. (Note that if R is an observable it must be an M_i, which is covered by the previous case.)

3. $(Q\ R)$ occurs outside all of the M_i. Subcases are:

(a) Q is a **lambda**-abstraction.

(b) Q is a primitive procedure.

We now prove the lemma for each of the cases:

1. $(Q\ R)$ is within one of the M_i, i.e., for some $j \in [1, m]$, M_j is an application and contains (or is) the redex $(Q\ R)$. Let $M_j = E_1[(Q\ R)]$. That is,

$$P = E_1[E_2[(Q\ R)]]$$

where

$$E_1[\ ] = C[M_1]_1 \ldots [\ ]_j \ldots [M_m]_m$$

and $E_1[E_2[\ ]] = E[\ ]$. By Lemma 2.2, $E_1[\ ]$ and $E_2[\ ]$ are evaluation contexts. On replacing the M_i by M_i', we get

$$P' = E_1'[M_j']$$

where $E_1'[\ ]$ is $E_1[\ ]$ with the M_i' replacing M_i. Since each M_i' is of the same kind as the corresponding M_i, $E_1'[\ ]$, like $E_1[\ ]$, is an evaluation context. Thus, condition 2a is satisfied.

2. $(Q\ R)$ contains some of the M_i. Subcases are:

   (a) Q is one of the M_i, i.e., for some $j \in [1, m]$, $Q = M_j$. Thus M_j is either a primitive procedural constant or a **lambda**-abstraction. Thus

   $$P = E[(M_j\ R)]$$

   On replacing M_i by M_i', we have

   $$P' = E'[(M_j'\ R')]$$

   Since each M_i is the same kind as M_i', $E'[\ ]$ is an evaluation context, and R' is a value. Thus, condition 2b is satisfied.

   (b) $Q = (\textbf{lambda}\ (x)\ Q_1)$, where Q_1 may contain some of the M_i. Stepping once,

   $$P = E[((\textbf{lambda}\ (x)\ Q_1)\ R)] \ \triangleright\ E[Q_1[x/R]]$$

   On replacing M_i by M_i', we have

   $$P' = E'[((\textbf{lambda}\ (x)\ Q_1')\ R')]$$

   where $E'[\ ]$ is an evaluation context and R' is a value. Stepping once, this gives

   $$P' \ \triangleright\ E'[Q_1'[x/R']]$$

   Now, since $E[Q_1[x/R]]$ takes one less step than P to converge to V, the pair $E[Q_1[x/R]]$ and $E'[Q_1'[x/R']]$ satisfies the inductive hypothesis. This implies that P and P' also satisfy the hypothesis.

   (c) Q is a primitive procedure f occurring outside all the M_i. Subcases are:

      i. R is an M_j. Stepping once

      $$P = E[(\mathsf{f}\ M_j)]$$

Replacing the M_i by M_i' we get

$$P' = E'[(\mathsf{f}\ M_j')]$$

where $E'[\ ]$ is an evaluation context. If M_j, and therefore M_j' are observables, P and P' satisfy condition 2c. If M_j' is a procedure value, then stepping once,

$$P = E[(\mathsf{f}\ M_j)] \ \triangleright E[U]$$
$$P' = E'[(\mathsf{f}\ M_j')] \ \triangleright E[U']$$

where $E'[\,]$ is an evaluation context like $E[\,]$. By Lemma 2.1, $U = U'$. Since $E[U]$ takes one less step than P to converge, the pair $E[U]$ and $E'[U]$ satisfy the inductive hypothesis, and so do P and P'.

ii. $R =$ (**lambda** (x) R_1), where R_1 may contain some M_i. Stepping once,

$$P = E[(\mathsf{f}\ R)] \ \triangleright E[U]$$

Replacing M_i by M_i', we get

$$P' = E'[(\mathsf{f}\ R')] \ \triangleright E'[U']$$

where $E'[\ ]$ is an evaluation context and R' is a procedural value. By Lemma 2.1, U and U' are identical. Clearly, $E[U]$ and $E'[U]$ satisfy the inductive hypothesis, since the former takes one less step to converge. Therefore, P and P' also satisfy the hypothesis.

3. $(Q\ R)$ occur outside all of the M_i. Subcases are:

   (a) Q is a **lambda**-abstraction. This case is treated exactly as in case 2b.

   (b) Q is a primitive procedure f Stepping once,

$$P = E[(\mathsf{f}\ R)] \ \triangleright E[U]$$

Replacing M_i by M_i', we have

$$P' = E'[(\mathsf{f}\ R)] \ \triangleright E[U]$$

Since $E[U]$ and $E'[U]$ must satisfy the inductive hypothesis, so too must P and P'. ∎

2.4 Control enhancements for PS

As we have seen, evaluating subterms in a PS program involves keeping track of the evaluation context. This is the textual context surrounding the current redex—in other words, the *rest of the computation* relative to the user. In a conventional machine implementation, a *control stack* encodes this information about the evaluation context. Control operators seize or alter this context, thereby allowing the user to manipulate the flow of control in a program.

The simplest act of control manipulation is to destroy or *abort all* of the evaluation context. This is equivalent to flushing the control stack. An enhanced approach limits the amount of evaluation context flushed. This is called *delimiting* the abort. In both cases, control can only be transferred to a dynamically enclosing context.

In contrast to the *first-order* control manipulation described above, *higher-order* control operators allow transfer of control to arbitrary points in the program without regard to dynamic extent. They provide the user with an abstraction of the current evaluation context, i.e., a "snapshot" of the current control stack. This abstraction, called the *continuation*, is usually a procedure, and may act in one of two ways when invoked: (1) it may reinstate the captured context *in place of* the invoking context; or (2) it may install the captured context *on top of* the invoking context. These variants are respectively called *abortive* and *functional* continuations.

2.5 First-order control manipulation

First-order control forms for aborting computation abound in real programming languages. They appear under various names, e.g., *error*, *stop*, *exit*, *halt*. In the simplest version, all of the evaluation context is flushed without returning any discriminating value. A more useful version—the one considered in detail here—has the syntax (**abort** M) and evaluates the subexpression (or operand) M *after* aborting the evaluation context. The stepping rule that captures this behavior is:

$$E[(\textbf{abort}\ M)] \ \triangleright M$$

The evaluation function for PS+**abort** is defined as for plain PS.

2.6 First-order delimiters

The **abort** operator described above removes *all* of the evaluation context. Many programming languages with first-order control usually also have a further facility that *delimits* the extent of context deleted by **abort**. One can view the delimiter as *receiving* or *handling* the aborted value. In this more general version, first-order control manipulation is that which allows computations to abort to any dynamically enclosing control context, not necessarily the topmost program context.

Examples of such controlled **abort**s are Lisp 1.5's *error* [34], Common Lisp's **throw** [49] and ML's **raise** [23, 37]. The corresponding control delimiters are respectively **errorset**, **catch** and **handle**. Typically, languages allow tags on the aborting and delimiting forms. For instance, in Common Lisp, a tagged **throw** can only be delimited by a **catch** with an identical tag. In other words, a **throw** can pick its destination, and not restrict itself to the *nearest* enclosing **catch**. While tagged operators are convenient, they will not be considered here, since they can be obtained as fairly straightforward—albeit inefficient—syntactic extensions from the untagged primitives.

It is no coincidence that many of the operators mentioned here are related to error or condition signalers and their handlers. Indeed, the condition systems of ML, Common Lisp, Dylan [11] and C++ [53] are sophisticated instances of first-order control usage.

In our idealized language, we call our delimiter the "prompt," %, recognizing that it treats its operand as an independent program, in so far as control actions are concerned [16, 51], much like the prompt sign in a read-eval-print loop.[1] The syntax for a prompt expression is $(\%\ M)$.

Evaluation contexts $E[\,]$ for the language PS+**abort**+% are described by a new grammar:

$$\begin{array}{rcl} E[\,] & ::= & [\,] \mid (E[\,]\ M) \mid (V\ E[\,]) \mid (\mathbf{abort}\ E[\,]) \mid (\%\ E[\,]) \mid (\%\ F[\,]) \\ F[\,] & ::= & [\,] \mid (F[\,]\ M) \mid (V\ F[\,]) \mid (\mathbf{abort}\ F[\,]) \end{array}$$

The stepping rules for **abort** and % are:

$$\begin{array}{rcl} E[(\%\ F[(\mathbf{abort}\ M)])] & \triangleright & E[M] \\ E[(\%\ V)] & \triangleright & E[V] \end{array}$$

showing that an **abort**-expression in the language PS+**abort**+% erases evaluation context only up to its *nearest* surrounding prompt. The evaluation function for PS+**abort**+% is defined as for plain PS and PS+**abort**.

2.7 Higher-order control manipulation

Higher-order control operators allow unrestricted transfers of control without regard to dynamic extent. An example is Scheme's and ML's *call-with-current-continua-*

[1]The symbol % is chosen for its similarity to an operating system prompt. Lisp's own prompt sign is usally >—unfortunately, that symbol is taken.

tion [5, 9, 54].[2] Like its historical forerunners **J** [31] and **escape** [41], *call/cc* is a *control reifier*: it *reifies* the control information of a running program and gives it to the user. The user can reinstate this control information at any time, thus creating arbitrary context jumps. The **goto** and labels of Algol-like languages provide a similar feature.[3]

A *call/cc* application, $(call/cc\ M)$, calls its subexpression M with an abstraction of its surrounding evaluation context, i.e., the "rest of the program" or the "current continuation"—hence the name of the operator. Invoking this continuation at any point in the program reinstates the captured context in place of the current one, and fills it with the continuation's argument. This ability to substitute the current context by a previously stored context is simple and powerful, as attested by the several elegant programming metaphors that it allows [10, 13, 22, 24, 26, 27, 46]. The stepping rule for *call/cc* is:

$$E[(call/cc\ M)] \ \triangleright E[(M\ (\mathbf{lambda}\ (v)\ (\mathbf{abort}\ E[v])))]$$

Thus, the reduction language supports **abort** expressions even if PS+*call/cc* itself does not. A syntactically cleaner approach would be to use *call/cc′*, a *call/cc* variant [20], as the control primitive, with the following stepping rule:[4]

$$E[(call/cc'\ M)] \ \triangleright (M\ (\mathbf{lambda}\ (v)\ (call/cc'\ (\mathbf{lambda}\ (d)\ E[v]))))$$

The operator *call/cc′* captures context like *call/cc* *and* aborts context like **abort**. Indeed, *call/cc′* applied to a vacuous application (i.e., thunk) is equivalent to an **abort**.

---

[2] Abbreviated *call/cc* or *cwcc* in Scheme and *callcc* in ML.

[3] The **goto** construct in the C language [30] provides only a limited form of *call/cc*'s power: it can manipulate only the control within a procedure body. A closer analog is the **setjmp/longjmp** pair in the C standard library.

[4] *call/cc′* is called $\mathcal{C}$ in [20]. Similarly, *fcontrol* (see below) is called $\mathcal{F}$. We use longer names for programming language tokens to avoid confusion with semantic symbols.

$$(\textbf{abort}\ M) \equiv (\mathit{call/cc}'\ (\textbf{lambda}\ (d)\ M))$$

Further, *call/cc′* can simulate *call/cc*:

$$\begin{aligned}\mathit{call/cc} \equiv (&\textbf{lambda}\ (r)\\ &\quad (\mathit{call/cc}'\ (\textbf{lambda}\ (k)\\ &\qquad\qquad (k\ (r\ k)))))\end{aligned}$$

illustrating that a *call/cc*-application includes an implicit application of its continuation.[5]

The operator *call/cc′*—like *call/cc*—furnishes *abortive* continuations. Although it is simpler than *call/cc* in not containing implicit continuation invocations, it is not simple where aborts are concerned: Both *call/cc′* and a *call/cc′*-based continuation abort when invoked. A variant of *call/cc′*, called *fcontrol* [20], resolves this complexity by providing *functional* continuations. The stepping rule for *fcontrol*-expressions is:

$$E[(\mathit{fcontrol}\ M)]\ \rhd (M\ (\textbf{lambda}\ (v)\ E[v]))$$

Thus, the continuation obtained through *fcontrol* does not include an implicit abort. In contrast to *call/cc*- and *call/cc′*-continuations, *fcontrol*-continuations are unremarkable from the ordinary procedures of PS—hence the description "functional." The lack of implicit aborts in the continuations is not a loss, since an *fcontrol*-application, like a *call/cc′*-application, always aborts its surrounding context. Thus,

[5]Unfortunately, this abbreviation does not optimize tail calls. A rigorous, if less clear, definition would be:

$$\begin{aligned}\mathit{call/cc} \equiv (&\textbf{lambda}\ (r)\\ &\ ((\mathit{call/cc}'\ (\textbf{lambda}\ (k)\\ &\qquad (k\ (\textbf{lambda}\ (d)\\ &\qquad\quad (r\ (\textbf{lambda}\ (v)\\ &\qquad\qquad (k\ (\textbf{lambda}\ (d)\ v))))))))\\ &\ \ 0))\end{aligned}$$

it is possible to get abortive continuations from functional continuations by simply wrapping an *fcontrol*-application around the continuation. E.g., the following describes *call/cc'* using *fcontrol*:

$$\begin{aligned}
&\mathit{call/cc'} \equiv (\textbf{lambda}\ (r) \\
&\qquad\qquad (\mathit{fcontrol}\ (\textbf{lambda}\ (k) \\
&\qquad\qquad\qquad\qquad (r\ (\textbf{lambda}\ (v)\ (\mathit{fcontrol}\ (\textbf{lambda}\ (d)\ (k\ \ v)))))))))
\end{aligned}$$

2.8 Higher-order delimiters

Real languages like Scheme and ML that feature the higher-order control operator *call/cc* include no corresponding higher-order delimiter. First, there is no need for a *first*-order prompt, since a *call/cc*-application and its continuation provides an ample substitute for a prompt and its **abort**.[6] Second, delimiters for true higher-order continuations can be created in an ad hoc fashion by employing additional higher-order continuations.[7] Keeping track of several such continuations with their respective quasi-delimiters requires sophisticated bookkeeping effort even for a single control metaphor [10, 26], not to mention simultaneous uses of different control abstractions. Thus it is useful to seek a higher-order generalization of the first-order prompt.

A straightforward extrapolation of the first-order delimiter suggests that higher-order prompts delimit the extent of evaluation context captured or erased by control operators and their continuations. Thus, a higher-order prompt [16] for *fcontrol* would use the following stepping rules:

[6] Indeed, this *call/cc*-translation works *too* well, since the "exit" it simulates works even after program control has gone past the delimiting context! This is usually not a problem, but if it is absolutely necessary to ensure failure when such exits are attempted, it is possible to constrain *call/cc* appropriately with some effort [26]. The continuations that the constrained *call/cc* issues are *first-order* continuations.

[7] See Chapter 7 for a Scheme ($\approx$ PS+*call/cc*+assignment) implementation of higher-order prompts that is necessarily inefficient. The implementation falls short of being true embedding—since it requires modifying the read-eval-print loop.

$$
\begin{aligned}
E[(\%\ F[(\mathit{fcontrol}\ M)])] &\ \triangleright\ E[(M\ (\mathbf{lambda}\ (x)\ F[x]))] \\
E[(\%\ V)] &\ \triangleright\ E[V]
\end{aligned}
$$

with the appropriate definitions for the evaluation contexts $E[\ ]$ and $F[\ ]$.

This is indeed a useful delimiter. Together with higher-order control reifiers like *call/cc*, *call/cc′* or *fcontrol*, the prompt supports a multitude of programming idioms [15, 46]. However, it does not completely generalize first-order **abort** and prompt. In the first-order scenario, an **abort** merely signals a control action, leaving it to the prompt to handle the control action by collecting the aborted value. This first-order separation of tasks suggests a similar separation for the higher-order case too. The control reifer should only create the control object or continuation, which the higher-order prompt delimits and handles.

A closer look at an operator such as *call/cc*—or *call/cc′* or *fcontrol*—reveals that it combines *two* actions: not only does it capture the current evaluation context as a continuation, it also invokes its argument, a procedure, on this continuation. In other words, the *handling* of the continuation takes place at the same site as the creation of the continuation. Taking our cue from the first-order control, we shift the site of continuation handling from the control reifier to the prompt. This drastically changes the aspect of both reifier and delimiter.

The new control-capturing operator is a stripped-down version of the *fcontrol* of the previous section—it needs no procedural argument to "receive" its continuation, since the delimiter takes care of control handling. The new delimiter, %, takes two subexpressions: (1) a computation that runs as a control-independent program; and (2) a dyadic procedure that will handle any control actions performed by the first subexpression. The evaluation contexts for PS+*fcontrol*+% are similar to those for PS+**abort**+%:

$$\begin{aligned} E[\,] \;&::=\; [\,] \mid (E[\,]\ M) \mid (V\ E[\,]) \mid (\textit{fcontrol}\ E[\,]) \\ &\qquad\qquad \mid (\%\ M\ E[\,]) \mid (\%\ E[\,]\ V) \mid (\%\ F[\,]\ V) \\ F[\,] \;&::=\; [\,] \mid (F[\,]\ M) \mid (V\ F[\,]) \mid (\textit{fcontrol}\ F[\,]) \end{aligned}$$

The stepping rules for *fcontrol* and % are:

$$\begin{aligned} E[(\%\ F[(\textit{fcontrol}\ V)]\ V')] \;&\rhd\; E[(V'\ V\ (\mathbf{lambda}\ (x)\ F[x]))] \\ E[(\%\ V\ V')] \;&\rhd\; E[V] \end{aligned}$$

In other words, an *fcontrol*-application throws its argument value and its delimited functional continuation to its surrounding prompt. The prompt invokes the handling procedure on them. If the body of the prompt's first subexpression did not cause any control actions, its value is returned normally. This comprehensive higher-order generalization of the simple **abort**/prompt mechanism provides an excellent tool for diverse forms of control programs [45].

2.9 The activity lemma for PS's control extensions

The activity lemma, Lemma 2.3, holds in identical form for the control extensions of PS too.

Proof For this, we note that in the preceding proof of the activity lemma, if program P is not a value, it can be decomposed into an evaluation context $E[\,]$ and a redex. For plain PS, the redex is the application of a procedure to a value. For extended PS, the redex can also be a control action.

For example, consider PS+**abort**.

In addition to the cases enumerated in the proof of Lemma 2.3, we have an additional subcase for cases 2 and 3. Both these new subcases can be treated identically. It is possible that

$$P = E[(\mathbf{abort}\ R)]$$

where R is any expression, possibly containing M_i. Stepping once,

$$P \ \triangleright R$$

Replacing M_i by M_i', we have

$$P' \ \triangleright R'$$

Since R takes one less step than P to converge, the inductive hypothesis holds for the pair R and R'. Hence, the hypothesis also holds for P and P'.

A similar proof holds for *all* the control extensions of PS. ∎

Chapter 3

Modeling PS

A denotational model for a language such as PS consists of one or more structures called *domains* and an interpretation or *meaning function* that maps terms from the language to meanings or *denotations* in the structures. The meaning function assigns meanings *compositionally*, i.e., the meaning of a composite term depends only on the meanings of its subterms. The model provides a mathematical description of the behavior of a program as a well-defined function of the behavior of the program's subterms.

This chapter studies the basic *direct* model for the simple (purely functional) call-by-value language PS.

3.1 Building models

We use Scott's *information systems* framework [43] to construct our models. For some basic background and notation about information systems, see Appendix A.

A domain is a collection of consistent, deductively closed sets of propositions. The elements in the domain are ordered by the subset relation: An element a is "below" an element b, $a \sqsubseteq b$, if the propositions describing a form a subset of the propositions describing b. A *compact* domain element is one that can be described as the deductive closure of a finite number of atomic propositions. I.e., it is an element that can be completely described by a finite number of propositions, even though, in general, infinitely many propositions may be true of it.

The compact elements of a domain form an easily described *and* representative subcollection of the entire domain. *Every* domain element can be described as the deductive closure of some combination of the domain's compact elements. In other words, a domain is isomorphic to the deductive (or "ideal") closure of its compact elements. Furthermore, many of the domain constructions, such as disjoint sums, function spaces, reflexive domains, etc., are obtained by defining the compact elements of the constructed domain in terms of the compact elements of the constituent subdomains. There is also a simple way to extend proofs that hold for the compact elements to all the domain elements—thus allowing us to concentrate on the compact elements. In short, the compact elements suffice to describe a domain completely.

3.2 The direct model for PS

The direct model for PS is a reflexive domain $\mathbf{D}$ that contains the integers and the strict functions on itself:

$$\mathbf{D} = \underbrace{\mathbf{O}}_{\text{integers}} \oplus \underbrace{[\mathbf{D} \to_s \mathbf{D}]_\perp}_{\text{procedures}}.$$

The appendix shows how to build such a domain. For convenience, we will use the tags inO (for *o*bservables) and inP (for *p*rocedures) instead of the customary inL (*l*eft) and inR (*r*ight). For example, the element for the number 5 is $(\mathsf{inO}(5))^\circ$; the procedural element that maps 1 to 1 (and everything else to $\perp$) is $(\mathsf{inP}\langle\{\mathsf{inO}(1)\},\{\mathsf{inO}(1)\}\rangle)^\circ$. This kind of notation rapidly becomes unwieldy for larger elements as the tags proliferate; we therefore omit the tags when there is no ambiguity. Thus, the two elements mentioned above are simply called 5 and $1 \Rightarrow 1$ respectively.

The function $\mathcal{A}$ (Figure 3.1) defines the meanings of PS terms. *Env* is the set of finite tables (environments) mapping variables to domain values (*other than* $\perp$). Meanings are defined compositionally, i.e., subterms contribute to a term's meaning only through their own meanings. The meaning of $\mathcal{A}'[\![P]\!]$ of a PS *program* P is simply $\mathcal{A}[\![P]\!]\bot$, where $\bot$ is the empty environment.

Domains

$$\mathbf{D} = \mathbf{O} \oplus (\mathbf{D} \to_s \mathbf{D})_\bot$$

Meaning functions

$$\mathcal{A}' : PS \to \mathbf{D}$$

$$\mathcal{A}'[\![P]\!] = \mathcal{A}[\![P]\!]\bot$$

$$\mathcal{A} : PS \to Env \to \mathbf{D}$$
$$Env : Var \to_f \mathbf{D} \setminus \{\bot\}$$

$$\begin{aligned}
\mathcal{A}[\![\ulcorner n \urcorner]\!]\rho &= \mathsf{inO}(n) \\
\mathcal{A}[\![add1]\!]\rho &= \mathsf{inP}(\lambda v.v = \mathsf{inO}(n) \hookrightarrow \mathsf{inO}(n+1), \bot) \\
\mathcal{A}[\![sub1]\!]\rho &= \mathsf{inP}(\lambda v.v = \mathsf{inO}(n) \hookrightarrow \mathsf{inO}(n-1), \bot) \\
\mathcal{A}[\![number?]\!]\rho &= \mathsf{inP}(\lambda v.v = \mathsf{inO}(n) \hookrightarrow \mathsf{inO}(1), \mathsf{inO}(0)) \\
\mathcal{A}[\![ef]\!]\rho &= \mathsf{inP}(\lambda v.v = \mathsf{inO}(0) \hookrightarrow \mathcal{A}[\![(\mathbf{lambda}\ (x\ y)\ y)]\!]\rho, \\
&\qquad\qquad\qquad\qquad\qquad \mathcal{A}[\![(\mathbf{lambda}\ (x\ y)\ x)]\!]\rho) \\
\mathcal{A}[\![x]\!]\rho &= \rho[x] \\
\mathcal{A}[\![(\mathbf{lambda}\ (x)\ M)]\!]\rho &= \mathsf{inP}(\lambda v.\mathcal{A}[\![M]\!]\rho[x/v]) \\
\mathcal{A}[\![(M\ N)]\!]\rho &= apply(\mathcal{A}[\![M]\!]\rho, \mathcal{A}[\![N]\!]\rho)
\end{aligned}$$

Auxiliary function

$$apply : \mathbf{D} \times \mathbf{D} \to_s \mathbf{D}$$

$$\begin{aligned}
apply(\mathsf{inP}(f), a) &= f(a) \\
apply(f, a) &= \bot \qquad \text{(for other values of } f\text{)}
\end{aligned}$$

Figure 3.1 The semantic function for PS.

3.3 Semantic equivalence relations

The crucial point about reasoning with the terms of a language is the ability to identify those terms that have the same "behavior." The model determines two natural equivalence relations on the closed terms in the language [36, 39]. First, there is the relation based on the meanings that the model provides for the respective terms:

Definition 3.1 *(Denotational Equivalence)* Two closed terms M and N are *denotationally equivalent*, $M \equiv N$, if their meanings are the same, i.e., $\mathcal{A}'[\![M]\!] = \mathcal{A}'[\![N]\!]$.

The second relation draws from the behavior of the terms as it appears to the programmer. The latter can witness a term's effect only by using it as a subterm in a program, and then observing the *program*'s behavior or meaning. In this view, two terms are equivalent if each can replace the other in any program whatsoever without affecting that program's behavior. For our purposes, it suffices to restrict attention to the program's *termination* behavior. This is because PS conditionals suffice to create contexts that produce differing termination behavior for terms that are inequivalent by any other observable measure.

Definition 3.2 *(Observational Equivalence)* Two closed terms M and N are *observationally equivalent*, $M \simeq N$, if for all contexts $C[\,]$ that are program contexts for M and N, $\mathcal{A}'[\![C[M]]\!] = \perp$ iff $\mathcal{A}'[\![C[N]]\!] = \perp$.

For example, $P_0 \equiv$ (**lambda** (f) Ω) and $P_1 \equiv$ (**lambda** (f) $((f\ 1)\ \Omega)$) are observationally equivalent: No program context can distinguish them. The proof requires the activity lemma (Lemma 2.3 of Chapter 2) and the following lemmas.

Lemma 3.1 If $P \equiv E[M]$ where $E[\,]$ is an evaluation context, then for all ρ,

$$\mathcal{A}[\![M]\!]\rho = \perp \quad \text{implies} \quad \mathcal{A}[\![E[M]]\!]\rho = \perp$$

Proof The proof is an induction on the structure of $E[\,]$.

- $(E[\,] = [\,].)$ The lemma is trivially satisfied.
- $(E[\,] = (V\ E'[\,]).)$ We have

  $$\mathcal{A}[\![(V\ E'[\,])]\!]\rho = apply(\mathcal{A}[\![V]\!]\rho, \mathcal{A}[\![E'[M]]\!]\rho)$$

  By induction,

  $$\mathcal{A}[\![M]\!]\rho = \bot \quad \text{implies} \quad \mathcal{A}[\![E'[M]]\!]\rho = \bot$$

  and from the definition of *apply*, this implies

  $$\mathcal{A}[\![E[M]]\!]\rho = \bot$$

- $(E[\,] = (E'[\,]\ M).)$ We have

  $$\mathcal{A}[\![(E'[\,]\ M)]\!]\rho = apply(\mathcal{A}[\![E'[M]]\!]\rho, \mathcal{A}[\![M]\!]\rho)$$

  By induction,

  $$\mathcal{A}[\![M]\!]\rho = \bot \quad \text{implies} \quad \mathcal{A}[\![E'[M]]\!]\rho = \bot$$

  and from the definition of *apply*, this implies

  $$\mathcal{A}[\![E[M]]\!]\rho = \bot \quad \blacksquare$$

Since the activity lemma is couched in terms of the reduction rules of Section 2.1, we need a lemma bridging reduction steps and the meaning function. In other words, reduction steps preserve denotations.

Lemma 3.2 If $M \xrightarrow{*} N$, then $\mathcal{A}[\![M]\!]\rho = \mathcal{A}[\![N]\!]\rho$ for all ρ.

Proof This is a well-known induction on the derivation $M \xrightarrow{*} N$ [4, 39]. The interesting step is:

$$\mathcal{A}[\![((\textbf{lambda}\ (x)\ M)\ V)]\!]\rho = \mathcal{A}[\![M[x/V]]\!]\rho$$

which is proved using the structure of M. ∎

In particular, the meaning of any program that converges is non-bottom, since, from the definition of the meaning function, the meaning of a value term is never bottom.

We can now *prove* our previous example terms M_0 and M_1 are indeed observationally indistinguishable.

Claim 3.1 The PS terms

$$\begin{array}{rcll} P_0 & \equiv & (\textbf{lambda}\ (f)\ \Omega) & \text{and} \\ P_1 & \equiv & (\textbf{lambda}\ (f)\ ((f\ 1)\ \Omega)) & \end{array}$$

are observationally equivalent.

Proof Consider any program context $C[\ ]$. Assume $C[P_0]$ converges. I.e., by Lemma 3.2, $\mathcal{A}'[\![C[P_0]]\!]$ is non-bottom. By the Activity Lemma, either $C[P_1]$ also converges, or for all procedural P,

$$C[P] \stackrel{*}{\triangleright} E[(P\ U)]$$

for some value U. In particular,

$$C[P_0] \stackrel{*}{\triangleright} E[(P_0\ U)]$$

By Lemma 3.2,

$$\mathcal{A}'[\![C[P_0]]\!] = \mathcal{A}'[\![E[(P_0\ U)]]\!]$$

By Lemma 3.1, since

$$\mathcal{A}[\![(P_0\ U)]\!]\bot = \bot$$

we have

$$\mathcal{A}'[\![C[P_0]]\!] = \perp$$

contradicting our assumption that $C[P_0]$ converges. Thus, $C[P_1]$ must converge.

By a similar argument, we can show that if $C[P_1]$ converges, so too must $C[P_0]$. Thus, P_0 and P_1 are observationally equivalent. ∎

In the following, we will often show that two terms M and N are observationally *in*equivalent by producing a program context $C[\,]$ such that the programs $C[M]$ and $C[N]$ map to two different numbers. By the definition above, we would need a context $C'[\,]$ such that only one of $C'[M]$ and $C'[N]$ has a non-bottom meaning. Fortunately, it is always easy to convert the "different numbers" context into the more orthodox one. For example, if the meaning function maps $C[M]$ and $C[N]$ to numbers 0 and 1 respectively, we can construct a context

$$C'[\,] = (\textbf{if } C[\,]\ 1\ \Omega)$$

such that $C'[M]$ diverges and $C'[N]$ converges.

Before we proceed with our investigations in the denotational domains, we need assurance that the model is *sound* with respect to the language. The assignment of meanings to the terms of the language should not result in any *observable* inconsistencies. In other words, if the model assigns the same meaning to two terms, then a program context filled with either term should behave the same. That is to say, denotationally equal terms should also be observably equal.

> **Theorem 3.1** *(Soundness)* For any two PS terms M and N, if $M \equiv N$, then $M \simeq N$.

Proof Owing to the compositional definition of $\mathcal{A}$, it follows that if $M \equiv N$, then $C[M] \equiv C[N]$ for all contexts, in particular program contexts. This means that M and N are observationally equivalent. ∎

The contrapositive reading is: Observationally *in*equivalent terms *must* have different denotations. This suggests an alternate view of soundness: An unsound model trivializes a language by collapsing obviously different terms to a single meaning.

The converse property, *full abstraction*, requires that observationally equivalent terms also be denotationally equivalent.

Definition 3.3 A model is *fully abstract* if for any two terms M and N, $M \simeq N$ implies $M \equiv N$.[8]

A model that ensures that observationally different terms map to differing meanings may nevertheless map observationally indistinguishable terms to different meanings. In other words, a model may be sound without being fully abstract. A sound model ensures that there are enough meanings so that observationally different terms do not map to the same meaning, whereas a fully abstract model guards against making too fine a distinction between terms than observation warrants.

Full abstraction typically fails for an otherwise sound model when the model has some capability that cannot be mimicked in the language. The model can thus distinguish more terms using this capability than the language itself can—thus, denotationally inequivalent terms can turn out to be observationally equivalent. Checks for full abstraction thus provide valuable information regarding *how* the programming language may be *enhanced*, viz., by suggesting an addition to the language that bridges the gap between model and language.

3.4 Achieving full abstraction

For the language PS, the direct model mentioned above is *not* fully abstract. The model can perform deterministically parallel disjunctions, whereas the language can-

[8] It is usual to combine the two directions of the implication in the definition of full abstraction. I.e., fully abstract models are those where $M \simeq N$ *iff* $M \equiv N$, or a fully abstract model is implicitly sound.

not. Plotkin [39] describes a similar failure of the call-by-name language PCF with respect to *its* model, by producing two language terms that are denotationally different but observationally indistinguishable. We can modify his counterexample to apply to PS too: consider the terms M_u $(u = 0, 1)$:

$$
\begin{array}{l}
M_u \stackrel{\mathrm{df}}{=} (\textbf{lambda}\ (x) \\
\qquad\quad (\textbf{if}\ (x\ (\textbf{lambda}\ (d)\ \Omega)\ (\textbf{lambda}\ (d)\ 1)) \\
\qquad\qquad (\textbf{if}\ (x\ (\textbf{lambda}\ (d)\ 1)\ (\textbf{lambda}\ (d)\ \Omega)) \\
\qquad\qquad\quad (\textbf{if}\ (x\ (\textbf{lambda}\ (d)\ 0)\ (\textbf{lambda}\ (d)\ 0)) \\
\qquad\qquad\qquad \Omega\ \ulcorner u \urcorner) \\
\qquad\qquad\quad \Omega) \\
\qquad\qquad \Omega))
\end{array}
$$

Claim 3.2 The terms M_0 and M_1 are observationally indistinguishable in PS.

Proof Consider a program context $C[\,]$ for the terms. Assume $C[M_0]$ converges. By the Activity Lemma, either $C[M_1]$ too converges, or for all procedural M,

$$C[M] \stackrel{*}{\triangleright} E[(M\ V)]$$

for some value term V. In particular,

$$C[M_u] \stackrel{*}{\triangleright} E[(M_u\ V)]$$

Assuming $\mathcal{A}[\![V]\!]\bot = v$, we have

$$
\mathcal{A}'[\![(M_u\ V)]\!] = \mathcal{A}\left[\begin{array}{l}
(\textbf{if}\ (x\ (\textbf{lambda}\ (d)\ \Omega)\ (\textbf{lambda}\ (d)\ 1)) \\
\quad (\textbf{if}\ (x\ (\textbf{lambda}\ (d)\ 1)\ (\textbf{lambda}\ (d)\ \Omega)) \\
\qquad (\textbf{if}\ (x\ (\textbf{lambda}\ (d)\ 0)\ (\textbf{lambda}\ (d)\ 0)) \\
\qquad\quad \Omega\ \ulcorner u \urcorner) \\
\qquad \Omega) \\
\quad \Omega)
\end{array}\right] \bot[x/v]
$$

There are two cases: The procedure v either (1) ignores both its arguments; (2) applies its first argument; or (3) applies its second argument. If the first, the **if**-tests always pick the then-branch, or always pick the else-branch. In either event, the meaning of $(M_u\ V)$ is bottom. In the second case, the first **if**-test produces bottom. In the last case, either the first **if**'s else-branch or the second **if**-test produces bottom. Thus, in all three cases:

$$\mathcal{A}[\![(M_u\ V)]\!]\bot = \bot$$

By Lemma 3.1, the meaning of $C[M_u]$ is bottom, contradicting the assumption that $C[M_0]$ converges. Therefore, $C[M_1]$ must converge.

Similarly, if $C[M_1]$ converges, so too does $C[M_0]$. Thus, M_0 and M_1 are observationally equivalent. ∎

Although we cannot find a PS context that distinguishes M_0 and M_1, the model for PS can tell the two terms apart. The model has the following function that performs parallel disjunction on thunks:

$$\underline{por} = \lambda m, n. \begin{cases} 0 & \text{if both } m \text{ and } n \text{ thaw to } 0 \\ \bot & \text{if both thaw to } \bot \\ 1 & \text{otherwise} \end{cases}$$

In particular, $\underline{por}$ can yield true (i.e., 1) as a return value even if one of its argument thunks diverges. Applying the denotations of M_0 and M_1 to $\underline{por}$ gives:

$$apply(\mathcal{A}[\![M_u]\!]\bot, \underline{por}) = u$$

In other words, M_0 and M_1 have different meanings, and hence $M_0 \not\equiv M_1$.

There is clearly a mismatch between the abilities of the language PS and those of its model. To rectify this, we can enhance PS with the conditional operator **pif**, which is similar to **if**, but can yield a result even when the test fails to converge. For convenience, we shall use the equivalent *procedure pif**, where:

$$(\textbf{pif}\ M\ \ N\ \ P) \stackrel{\text{df}}{=}$$
$$(\mathit{pif}^*\ (\textbf{lambda}\ (d)\ M)\ (\textbf{lambda}\ (d)\ N)\ (\textbf{lambda}\ (d)\ P))$$

Setting the informal description above in denotational terms, we arrive at the denotational semantics of *pif**:

$$\mathcal{A}[\![\mathit{pif}*]\!]\rho = \underline{\mathit{pif}^*}$$

where $\underline{\mathit{pif}^*}$ is the function:

$$\lambda b, t, e. \begin{cases} \mathit{thaw}(e) & \text{if } \mathit{thaw}(b) \text{ is } 0 \\ \mathit{thaw}(t) & \text{if it is neither } 0 \text{ nor } \perp \\ n & \text{if } \mathit{thaw}(t) = \mathit{thaw}(e) = n\text{, a number} \\ \lambda v.\underline{\mathit{pif}^*}\ b\ (\mathit{freeze}((\mathit{thaw}(t))v))\ (\mathit{freeze}((\mathit{thaw}(e))v)) & \\ & \text{if } \mathit{thaw}(t) \text{ and } \mathit{thaw}(e) \text{ are both procedures} \end{cases}$$

Note that unlike the typed languages PCF [39] and value-PCF [47], the PS form of **pif** has to deal with procedures in the branches of the conditional. In this case, the selection of the branch is dependent on the eventual application of the entire **pif**-expression.

The parallel-**or** on thunks, *por-th*, is defined as:

$$\mathit{por\text{-}th} \stackrel{\text{df}}{=} (\textbf{lambda}\ (m\ \ n)$$
$$(\textbf{pif}\ (m\ \ 0)\ 1\ (\textbf{pif}\ (n\ \ 0)\ 1\ 0))))$$

Now, using the context ([ ] *por-th*), the terms M_0 and M_1 are indeed observationally different.

3.5 Proving full abstraction

To prove that the model is fully abstract for PS+**pif**, we follow a modification of the proof for PCF [39]. We first prove a lemma that all the *compact* elements in the model are *definable* in PS, i.e., each compact element is the meaning of some term in the model. Using this result, we show that denotationally inequivalent terms cannot be observationally equivalent.

To aid the inductive proof, the lemma states that additional procedures mapping the compact elements to boolean values (here $0, 1$) are also definable, in addition to the compact elements themselves.

Lemma 3.3 For all compact elements e and f in $\mathbf{D}$:

1. e is definable;
2. if e is non-bottom, then $e \Rightarrow 1$, the procedure mapping e to 1, is definable; and
3. if e and f are inconsistent, then $e \Rightarrow 1 \sqcup f \Rightarrow 0$, the procedure mapping e to 1 and f to 0, is definable.

Proof The proof is an induction on the *size* of the compact elements. The size of a compact element is the sum of the sizes of the precompact elements forming its minimal representation. For observable (i.e., non-procedural) domains, the size of a precompact element is one. For procedural domains, the size of a precompact element is the sum of the sizes of its argument (which is a compact element) and its result (which is a precompact element).

1. Just e. e can be either bottom, a number, or a procedure.
   (a) (e is bottom.) The defining term is Ω.
   (b) (e is a number n.) The defining term is $\ulcorner n \urcorner$.
   (c) (e is a procedure.) Let $E = \{\ldots p_r \ldots\}$ be *a* minimal representation for e, where the p_r are precompact elements. Each p_r is of the form $a_{r,0} \Rightarrow \ldots a_{r,k_r} \Rightarrow b_r$, with b_r an observable. There are two cases.

i. First, assume that for each m, the various $a_{r,m}$ are consistent. First, this means that all the precompact elements in E are of the same length (say k)—since otherwise we would have an element $a_{i,0} \Rightarrow \ldots a_{i,k_i} \Rightarrow b_i$ and a *longer* element $a_{j,0} \Rightarrow \ldots a_{j,k_i} \Rightarrow \ldots a_{j,k_j} \Rightarrow b_j$, where the pairs $a_{i,m}, a_{j,m}$ are consistent for $m = 0$ through k_i, but b_i and $a_{j,k_{i+1}} \Rightarrow \ldots aj, k_j \Rightarrow b_j$ are inconsistent. Second, since the $a_{r,m}$ are consistent for each m, all the b_r are consistent too, and since they are observables, they must also be the *same* observable, say b.[9]

Consider one such element $p_i \equiv a_{i,0} \Rightarrow \ldots a_{i,k} \Rightarrow b$. By induction, each $a_{i,n} \Rightarrow 1$ (n varies) is definable by an *An?* say, b by B say, and $P_i' = E \setminus \{p_i\}$ is definable by *Mi* say. Then, e itself is definable by:

(**lambda** ($x_0 \ldots x_k$)
  (**pif** (**and** ... (*An?* x_n) ...) B
    (*Mi* ... x_n ...)))

ii. If the previous assumption is false, then there elements p_i and p_j in E such that $a_{i,m}$ and $a_{j,m}$ are inconsistent. Find the smallest such m. Several things can now be noted.

A. Neither $a_{i,m}$ nor $a_{j,m}$ can be bottom—since bottom is consistent with everything.

B. For every l before m, the $a_{r,l}$ (where r varies) must all be consistent —since m is the smallest index where inconsistency crops up.

C. m cannot be larger than the smallest k_r (say k_u)—since otherwise we would have two elements $p_u \equiv a_{u,0} \Rightarrow \ldots a_{u,k_u} \Rightarrow b_u$ and

[9]It could be argued that only some of the b_i need be the (same) non-bottom observable—the other b_i could well be bottom. However, this case cannot arise if E were a minimal representation: None of the p_r with $b_r = \perp$ need be in E, as they do not add any information. The case where *all* the b_r are bottom does not pose any problem, though it may be noted that the set E, if it is a minimal representation, must then be a singleton.

$p_v \equiv a_{v,0} \Rightarrow \ldots a_{v,k_u} \Rightarrow \ldots a_{v,k_v} \Rightarrow b_v$ where the pairs $a_{u,s}, a_{v,s}$ are consistent for $s = 0$ through k_u, but b_u and $a_{v,k_{u+1}} \Rightarrow \ldots a_{v,k_v} \Rightarrow b_v$ are inconsistent—the former is an observable and the latter a procedure. This would create the impossible situation of a procedure that takes consistent inputs to inconsistent results.

D. From the above, e must be a (curried) procedure of at least $m+1$ arguments.

Since $a_{i,m}$ and $a_{j,m}$ are inconsistent, $a_{i,m} \Rightarrow 1$ and $a_{j,m} \Rightarrow 0$ are consistent. Thus, $a_{i,m} \Rightarrow 1 \sqcup a_{j,m} \Rightarrow 0$ exists, and by induction, is definable. Let this defining term be *Aij?*.

Let $E'_i = E \setminus \{p_i\}$ and $E'_j = E \setminus \{p_j\}$. Again, by induction, these (i.e., E'°_i and E'°_j) are definable, by *Mi* and *Mj*, say.

Therefore, $e = E^{\circ}$ itself is definable by:

(**lambda** (x_0 … x_m)
  (**pif** (*Aij?* x_m) (*Mj* x_0 … x_m) (*Mi* x_0 … x_m)))

2. $e \Rightarrow 1$, procedure mapping e to 1, where e is either a number or a procedure, but not bottom.

(a) (e is a number n.) Define the predicates $\ulcorner n \urcorner ?$ as follows: For a positive n,

$$\ulcorner n \urcorner ? \stackrel{\text{df}}{=} (\textbf{lambda}\ (x)\ (\textit{not}\ (\textit{sub1}^n\ x)))$$

where $\textit{sub1}^n$ is the n-fold composition of the procedure *sub1*. The procedure *not* for testing zerohood is:

$$\textit{not} \stackrel{\text{df}}{=} (\textbf{lambda}\ (x)\ (\textbf{if}\ x\ 1\ 0))$$

If n is negative, i.e., $n = -m$ for positive m, then the predicate is:

$$\ulcorner n \urcorner ? \stackrel{\text{df}}{=} (\textbf{lambda}\ (x)\ (\textit{not}\ (\textit{add1}^m\ x))))$$

Then, $n \Rightarrow 1$ is defined by:

$$(\textbf{lambda}\ (x)\ (\textbf{if}\ (\textbf{and}\ (\textit{number?}\ x)\ (\ulcorner n \urcorner ?\ x))\ 1\ \Omega))$$

(b) (e is a procedure.) Let $E = \{\ldots a_i \Rightarrow b_i \ldots\}$. Let Ai define a_i and $Bi?$ define $b_i \Rightarrow 1$. Then e is defined by:

$$\begin{array}{l} (\textbf{lambda}\ (f) \\ \quad (\textbf{if}\ (\textbf{and}\ (\textit{procedure?}\ f)\ \ldots\ (Bi?\ (f\ Ai))\ \ldots)\ 1\ \Omega)) \end{array}$$

3. $e \Rightarrow 1 \sqcup f \Rightarrow 0$, procedure mapping e to 1 and f to 0, where e and f are inconsistent. There are three cases: Both e and f are numbers; only one of them is a number; both are procedures.

(a) (Both are numbers.) Let e and f be the numbers m and n respectively. Clearly, $m \neq n$.

$$\begin{array}{l} (\textbf{lambda}\ (x) \\ \quad (\textbf{if}\ (\textit{number?}\ x) \\ \quad\quad (\textbf{if}\ (m?\ x)\ 1 \\ \quad\quad\quad (\textbf{if}\ (n?\ x)\ 0\ \Omega)) \\ \quad\quad \Omega)) \end{array}$$

(b) (Only one of them, say e, is a number.) Let e be the number m. Let $F?$ define $f \Rightarrow 1$. Then, the defining term is:

$$\begin{array}{l} (\textbf{lambda}\ (x) \\ \quad (\textbf{if}\ (\textbf{and}\ (\textit{number?}\ x)\ (m?\ x))\ 1 \\ \quad\quad (\textbf{if}\ (\textbf{and}\ (\textit{procedure?}\ x)\ (F?\ x)) \\ \quad\quad\quad 0\ \Omega))) \end{array}$$

(c) (Both are procedures.) Then, there is a $p = p_0 \Rightarrow p_1$ in E and $q = q_0 \Rightarrow q_1$ in F such that p and q are inconsistent. I.e., p_0 and q_0 are consistent, but p_1 and q_1 are not.

Let R define $p_0 \sqcup q_0$. Let $PQ?$ define $p_1 \Rightarrow 1 \sqcup q_1 \Rightarrow 0$. Let $E?$ and $F?$ define $e \Rightarrow 1$ and $f \Rightarrow 1$ respectively. Then the defining term is:

$$
\begin{array}{l}
(\textbf{lambda}\ (f) \\
\quad (\textbf{if}\ (\mathit{not}\ (\mathit{procedure?}\ f))\ \Omega \\
\quad\quad (\textbf{if}\ (PQ?\ (f\ R))\ (E?\ f) \\
\quad\quad\quad (\mathit{not}\ (F?\ f)))))
\end{array}
$$

The proof of full abstraction of the model follows from the lemma.

Theorem 3.2 The direct model is fully abstract for PS+**pif**.

Proof We need to show that for any two closed terms M and N, if $M \simeq N$, then $M \equiv N$. Assume $M \not\equiv N$, i.e., $\mathcal{A}'[\![M]\!] = m$ and $\mathcal{A}'[\![N]\!] = n$ where m and n are different domain elements. We now need to show that no context $C[\,]$ can distinguish M and N by termination behavior.

The difference in m and n may consist in the two elements belonging to different partitions of the domain, where the partitions are bottom, observables and procedures —or they may be different elements in the same partition. We shall first distinguish values from different partitions, and then within each partition.

1. Distinguishing bottom from the rest: Suppose m is bottom, and n some non-bottom value. Then the empty context $[\,]$ distinguishes M and N.

2. Distinguishing observables (integers) from procedures: Let m be a number and n a procedure. Then the context $(\mathit{number}?\ [\,])$ distinguishes the terms.

3. Distinguishing between different numbers: Simple arithmetic operations ($\ulcorner n \urcorner$*?*) distinguish the terms.

4. Distinguishing between different procedures: Since they are not equal, there must be a precompact element p in one of them—say m—that is not in the other. Let p be $a_0 \Rightarrow \ldots a_k \Rightarrow b$, where the a_i are compact and b is either bottom or an integer. By Lemma 3.3, the a_i are definable through *Ai*, and b through B, which is either Ω or a numeral. A context that applies the terms to *A0*, ..., *Ak*, gives B in the case of M, and something else for N, which can then be distinguished by a further surrounding context as above. ∎

Chapter 4

The continuation model

When modeling languages with control operators, the customary tool is the continuation model pioneered by Strachey and Wadsworth [52]. This model is also called the *continuation-passing-style* (or simply: cps) model, since its semantic function uses an additional "continuation" argument. At any point, the continuation argument represents the rest of the computation. In essence, the model encodes a stack machine for the language.

The value of the model consists in suggesting that the model's continuations can also be provided at the level of the language itself, thereby creating a more powerful facility for nonlocal control manipulation than untagged or tagged **abort**. Indeed, the introduction of higher-order control operators such as **J** [31], **escape** [41] and **catch** or *call/cc* [54] is a recognition of this idea from abstract machines like the continuation model. This chapter will show that there is more advice to be gleaned from the continuation model: It suggests not only control-*packaging* operators like *call/cc* but also control-*delimiting* mechanisms.

4.1 The cps model for PS

Like the basic direct model (Chapter 3), the cps domain is the *sum* of two subdomains, one for observables and the other for procedures:

$$\begin{aligned} \mathbf{D} &= \overbrace{\mathbf{O}}^{\text{integers}} \oplus \overbrace{[\mathbf{D} \to_s \mathbf{C} \to_s \mathbf{D}]_\perp}^{\text{procedures}} \\ \mathbf{C} &= \underbrace{[\mathbf{D} \to_s \mathbf{D}]_\perp}_{\text{continuations}} \end{aligned}$$

Once again, the tags inO and inP identify elements in the observable and procedure subdomains respectively. To lessen notational clutter, the tags may be dropped when they are obvious from context.

Unlike the direct model, the procedure subdomain is not a simple function space; instead, each procedure denotation is a map from the argument denotation and a denotation representing the rest of the computation—a *continuation* function—to an answer. This scheme accounts for control operations in the procedure's body that could affect the rest of the computation.

The function $\mathcal{C}$ (Figure 4.1) defines the meanings of PS terms in the cps model. Each meaning is a procedure that takes an environment as well as a continuation argument. *Env* is the set of finite maps or environments mapping variables to non-bottom values in $\mathbf{D}$. The meaning of a program P is the meaning of the term P applied to the empty environment $\perp$ and the empty continuation $(\lambda v.v)$.

$$\begin{array}{rcl}\mathbf{D} & = & \mathbf{O} \oplus [\mathbf{D} \rightarrow_s \mathbf{C} \rightarrow_s \mathbf{D}]_\perp \\ \mathbf{C} & = & [\mathbf{D} \rightarrow_s \mathbf{D}]_\perp\end{array}$$

$$\begin{array}{rcl}\mathcal{C}' & : & \mathit{Programs} \rightarrow \mathbf{D} \\ \mathcal{C}'[\![M]\!] & = & \mathcal{C}[\![M]\!]\perp(\lambda v.v)\end{array}$$

$$\mathit{Env} : \mathit{Var} \rightarrow_f \mathbf{D} \setminus \{\perp\}$$

$$\begin{array}{rcl}\mathcal{C} & : & \mathit{Terms} \rightarrow \mathit{Env} \rightarrow \mathbf{C} \rightarrow_s \mathbf{D} \\ \mathcal{C}[\![\mathsf{n}]\!]\rho\kappa & = & \kappa(\mathsf{inO}(n)) \\ \mathcal{C}[\![\mathsf{f}]\!]\rho\kappa & = & \kappa(\mathsf{inP}(f)) \\ \mathcal{C}[\![x]\!]\rho\kappa & = & \kappa(\rho[x]) \\ \mathcal{C}[\![(\mathbf{lambda}\ (x)\ M)]\!]\rho\kappa & = & \kappa(\mathsf{inP}(\lambda v : \mathbf{D}, \kappa' : \mathbf{C}.\mathcal{C}[\![M]\!]\rho[x/v]\kappa')) \\ \mathcal{C}[\![(M\ N)]\!]\rho\kappa & = & \mathcal{C}[\![M]\!]\rho(\lambda m.\mathcal{C}[\![N]\!]\rho(\lambda n.mn\kappa))\end{array}$$

Figure 4.1 The cps model for PS.

4.2 Full abstraction of the cps model

The cps model is adequate for PS, indeed for PS+**abort**, PS+*call/cc*, and PS+*call/cc'*. However, it is not fully abstract for any of these languages. As in the direct model for PS, the model supports parallel disjunction. However, the addition of a **pif** operator is not enough to retrieve full abstraction: the language's control operators also demand a control-related extension.

Claim 4.1 The cps model is not fully abstract for PS.

Proof

1. Recall $P_0 = (\textbf{lambda}\ (f)\ \Omega)$ and $P_1 = (\textbf{lambda}\ (f)\ ((f\ 1)\ \Omega))$ from Claim 3.1. Using a similar proof, these terms are observationally equivalent in PS with the meanings provided by $\mathbf{D}_c$. The cps model, however, can distinguish them, using

   $$\kappa_p = \lambda m.m(\lambda v, \kappa.v)(\lambda x.x)$$

   For $\mathcal{C}[\![P_u]\!]\bot\kappa_p$ produces $\bot$ and 0 for $u = 0$ and 1 respectively.

2. Like the direct models, the cps model also supports parallel disjunction. Consider the terms M_0 and M_1 of Claim 3.2:

$$
\begin{aligned}
M_u \overset{\text{df}}{=} (&\textbf{lambda}\ (x)\\
&\quad(\textbf{if}\ (x\ (\textbf{lambda}\ (d)\ 1)\ (\textbf{lambda}\ (d)\ \Omega))\\
&\qquad(\textbf{if}\ (x\ (\textbf{lambda}\ (d)\ \Omega)\ (\textbf{lambda}\ (d)\ 1))\\
&\qquad\quad(\textbf{if}\ (x\ (\textbf{lambda}\ (d)\ 0)\ (\textbf{lambda}\ (d)\ 0))\\
&\qquad\qquad\Omega\ \ulcorner u \urcorner)\\
&\qquad\quad\Omega)\\
&\qquad\Omega))
\end{aligned}
$$

M_0 and M_1 are denotationally different in the cps model, since $\mathcal{C}[\![M_u]\!]\bot\kappa_m = u$, with the following definition for κ_m:

$$\kappa_m = (\lambda m.m\underline{por}(\lambda x.x))$$

where $\underline{por}$ is a function that performs parallel disjunction on thunks:

$$\underline{por} = \lambda m,\kappa.\kappa\lambda n,\kappa.\begin{cases} \kappa 0 & \text{if } (m0\lambda x.x) \text{ and } (n0\lambda x.x) \text{ are both } 0 \\ \bot & \text{if } (m0\lambda x.x) \text{ and } (n0\lambda x.x) \text{ are both } \bot \\ \kappa 1 & \text{otherwise} \end{cases}$$

However, M_0 and M_1 are observationally equivalent in PS, as we have already seen. ∎

In the first counterexample, the continuation κ_p is clearly the equivalent of an *aborting* action. The second counterexample shows that the cps model can express deterministic parallelism, just like the other models. Adding a control extension such as **abort** to the language will eliminate the first problem. The cps semantic clause for **abort**-expressions is:

$$\mathcal{C}[\![(\textbf{abort } M)]\!]\rho\kappa = \mathcal{C}[\![M]\!]\rho(\lambda v.v)$$

I.e., the current continuation, or the rest of the computation is thrown away and supplanted by the empty or identity continuation.

Even with parallel disjunction, the **abort** is not enough to secure full abstraction—the cps model supports far more elaborate control constructs. However, no amount of enhancement to the language via control extensions can eliminate the second counterexample, since control manipulation are not enough to escape divergence.

We shall concentrate first on *control-specific* deficiencies in the language vis-a-vis the cps model. But, first, we reformulate Lemma 3.1 to suit the cps model.

Lemma 4.1 If $P \equiv E[M]$ where $E[\,]$ is an evaluation context, then for all ρ and κ, there is a κ' such that:

$$\mathcal{C}[\![P]\!]\rho\kappa = \mathcal{C}[\![M]\!]\rho\kappa'$$

Proof The proof is an induction on the structure of $E[\,]$.

- ($E[\,] = [\,]$.) The lemma is trivially satisfied.
- ($E[\,] = (V\ E'[\,])$.) We have

$$\begin{array}{lll} & \mathcal{C}[\![(V\ E'[M])]\!]\rho\kappa & \\ = & \mathcal{C}[\![V]\!]\rho(\lambda m.\mathcal{C}[\![E'[M]]\!]\rho\lambda n.mn\kappa) & \\ = & \mathcal{C}[\![E'[M]]\!]\rho\lambda n.m_0n\kappa & \text{for some } m_0, \text{ since } V \text{ is a value term} \\ = & \mathcal{C}[\![M]\!]\rho\kappa' & \text{for some } \kappa', \text{ by induction} \end{array}$$

- ($E[\,] = (E'[\,]\ N)$.) We have

$$\begin{array}{lll} & \mathcal{C}[\![(E'[M]\ N)]\!]\rho\kappa & \\ = & \mathcal{C}[\![E'[M]]\!]\rho\kappa'' & \text{for some } \kappa'' \\ = & \mathcal{C}[\![M]\!]\rho\kappa' & \text{for some } \kappa', \text{ by induction} \end{array}$$

We now return to the question of full abstraction of the cps model.

Claim 4.2 There is a control-specific counterexample to full abstraction of the cps model for PS+**abort**.

Proof Consider the terms N_u $(u = 0, 1)$:

$$N_u \stackrel{\mathrm{df}}{\equiv} (\textbf{lambda}\ (p)$$
$$(\textit{equal?}\ (p\ (\textbf{lambda}\ (c)\ ((c\ 1)\ \Omega)))$$
$$(p\ (\textbf{lambda}\ (c)\ ((c\ \ulcorner u \urcorner)\ \Omega)))))$$

Let $C[\ ]$ be a program context. If $C[N_0]$ converges—i.e., $\mathcal{C}'[\![C[N_0]]\!] \neq \bot$—, then, by the activity lemma (Lemma 2.3), the program $C[N_1]$ also converges, *or*, for all procedural N,

$$C[N] \stackrel{*}{\triangleright} E[(N\ V)]$$

for some value V. From Lemma 4.1 and 3.2

$$\mathcal{C}'[\![C[N]]\!] = \mathcal{C}[\![(N\ V)]\!]\bot\kappa'$$

for some κ'. In the case of N_u, this meaning is

$$\mathcal{C}\left[\begin{array}{l} (\textit{equal?}\ (p\ (\textbf{lambda}\ (c)\ ((c\ 1)\ \Omega))) \\ \qquad\qquad (p\ (\textbf{lambda}\ (c)\ ((c\ \ulcorner u \urcorner)\ \Omega)))) \end{array}\right]\bot[p/v]\kappa'$$

for some procedure denotation v for V.

The first v-application may either ignore, return, abort or apply its argument. In the first three cases, the entire denotation is independent of u. In the final case, v's argument itself may either ignore, return, apply or abort *its* argument. In the first three cases, the entire denotation is bottom. In the final case, the entire denotation is 1, irrespective of u. Thus, N_0 and N_1 are observationally equivalent in PS+**abort**.

However, $N_0 \not\equiv N_1$ in the cps model. For, taking

$$\kappa_n = \lambda m.m(\lambda f, \kappa.f(\hat{\jmath}\,\kappa)(\lambda v.v))$$

where

$$\hat{\jmath}\,\kappa = \lambda v, \kappa'.\kappa v$$

we have $C[\![N_u]\!]\bot\kappa_n = u$. ∎

The semantic function ĵ produces a continuation flipper from a continuation argument—the function $\hat{\jmath}\,\kappa = \lambda v, \kappa'.\kappa v$ corresponds to a procedural value that substitutes one continuation (ĵ's argument) for another (the current continuation of the procedure). There is no expression in PS+**abort** that has this denotation. On the other hand, this is exactly the property of continuation objects provided by the operator *call/cc* (Chapter 2). For example, adding *call/cc* to PS requires the following meaning clause:

$$\mathcal{C}[\![(\mathit{call/cc}\ M)]\!]\rho\kappa = \mathcal{C}[\![M]\!]\rho(\lambda m.m(\hat{\jmath}\,\kappa)\kappa)$$

Or, we could use *call/cc'*, which combines the actions of **abort** and *call/cc*:

$$\mathcal{C}[\![(\mathit{call/cc'}\ M)]\!]\rho\kappa = \mathcal{C}[\![M]\!]\rho(\lambda m.m(\hat{\jmath}\,\kappa)(\lambda v.v))$$

In PS+*call/cc*, the context ([ ] *call/cc*) distinguishes N_0 and N_1. (In PS+*call/cc'*, the distinguishing context is ([ ] *call/cc'*).) Thus, both languages eliminate this particular counterexample to the model's full abstraction.[10] Still, even this control enhancement does not suffice to eliminate all control-specific violations of full abstraction.

> **Claim 4.3** There is a control-specific counterexample to full abstraction of the cps model for PS+*call/cc'*.

Proof Consider the terms Q_u $(u = 0, 1)$:

$$\begin{aligned} Q_u \stackrel{\text{df}}{=} (&\textbf{lambda}\ (z) \\ &\quad (\mathit{equal?}\ (z\ (\textbf{lambda}\ (d)\ (\textbf{abort}\ 1))) \\ &\qquad\quad (z\ (\textbf{lambda}\ (d)\ (\textbf{abort}\ \ulcorner u \urcorner)))))) \end{aligned}$$

[10]Since *call/cc'* can express **abort**, it is not necessary to retain **abort**. In the following, (**abort** M) in PS+*call/cc'* is an abbreviation for (*call/cc'* (**lambda** (d) M)), where d is not free in M.

Let $C[\ ]$ be a program context, and let $C[Q_0]$ converge. By activity, $C[Q_1]$ also converges, *or*, for all procedural Q,

$$C[Q] \overset{*}{\triangleright} E[(Q\ V)]$$

for some value V. From Lemma 4.1, the meaning of the program is

$$\mathcal{C}[\![(Q\ V)]\!]\bot\kappa'$$

for some κ'. Expanding, this denotation is:

$$\mathcal{C}\left[\begin{array}{l} (\mathit{equal?}\ (z\ (\mathbf{lambda}\ (d)\ (\mathbf{abort}\ 1))) \\ \qquad\qquad (z\ (\mathbf{lambda}\ (d)\ (\mathbf{abort}\ \ulcorner u \urcorner)))) \end{array}\right]\bot[z/v]\kappa'$$

for some procedure denotation v for V.

If v does not apply its argument, the program denotation cannot depend on u. If v does apply its argument, the first z-application and the behavior of **abort** ensure the program's meaning is 1, again independent of u. Thus, $Q_0 \simeq Q_1$ in PS+*call/cc'*.

On the other hand, with the continuation

$$\kappa_q = \lambda m.m[\lambda f, \kappa.\kappa(f0(\lambda v.v))](\lambda v.v)$$

we have $\mathcal{C}[\![Q_u]\!]\bot\kappa_q = u$, and hence $Q_0 \not\cong Q_1$. ∎

The continuation k_q in the above proof first runs the thunk f in an identity continuation before passing the result on to k, the continuation calling f. This makes the result of calling f available for immediate examination, even if it involved a nonlocal jump. In other words, the model can run a subcomputation as an *independent* program whose control actions are isolated from the enclosing program. When the subcomputation finishes, its result is used to resume the rest of the computation. PS+*call/cc'* cannot simulate this control-*delimiting* action since aborts and nonlocal jumps are irrevocable.

To fix this problem, we augment PS+*call/cc*$'$ with a control delimiter or *prompt* (Chapter 2). In this setting, a prompt-expression delimits all control actions—context captures *and* switches—that it dynamically encloses. The clause that extends the semantic functional $\mathcal{C}$ to include prompt-expressions is:

$$\mathcal{C}[\![(\%M)]\!]\rho\kappa = \kappa(\mathcal{C}[\![M]\!]\rho(\lambda v.v))$$

The operator that eliminates the second counterexample in the proof of Claim 4.1 is once again a parallel disjunction *por* based on **pif**, whose behavior was discussed in Chapter 3. The $\mathcal{C}$-clause for **pif**-expressions is:

$$\mathcal{C}[\![(\mathbf{pif}\ M\ N\ P)]\!]\rho\kappa = \begin{cases} \mathcal{C}[\![N]\!]\rho\kappa & \text{if } \mathcal{C}[\![M]\!]\rho\lambda v.v > 0 \\ \mathcal{C}[\![P]\!]\rho\kappa & \text{if } \mathcal{C}[\![M]\!]\rho\lambda v.v = 0 \\ \mathcal{C}[\![N]\!]\rho\kappa & \text{if } \mathcal{C}[\![N]\!]\rho\kappa = \mathcal{C}[\![P]\!]\rho\kappa \\ \bot & \text{otherwise} \end{cases}$$

Augmenting PS+*call/cc*$'$ with **pif** and prompt, we get a language for which the cps model is fully abstract. In particular, the above counterexample is dispelled by using the context ([ ] *por-th*) to distinguish M_0 and M_1. The procedure *por-th*, defined exactly as in Chapter 3, performs a paralel disjunction on its argument thunks. To prove full abstraction of the cps model for PS+*call/cc*$'$+**pif**+%, we need do show that compact elements are definable. The approach differs from that of Chapter 3 in that we need *two* notions of definability, for we now have complex values from two constructed domains, i.e., **D** and **C**.

Definition 4.1 *(Definability)*

1. **D**-*definability:* An element v in **D** is **D**-definable if there is a term M such that for all ρ and κ:

$$C[\![M]\!]\rho\kappa = \begin{cases} \bot & \text{if } v = \bot \\ \kappa v & \text{otherwise} \end{cases}$$

2. **C**-*definability:* An element κ in **C** is **C**-definable if there is a context $E[\,]$ such that for all M and ρ:

$$\mathcal{C}[\![E[M]]\!]\rho\lambda v.v = \mathcal{C}[\![M]\!]\rho\kappa$$

Lemma 4.2 For all compact elements e, f in **D**, and κ, γ in **C**:

1. e is **D**-definable;
2. if e is not bottom, then $e \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1$, the procedure mapping e to 1, is **D**-definable;
3. if e and f are inconsistent, then $[e \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1] \sqcup [f \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 0]$, the procedure mapping e to 1 and f to 0, is **D**-definable;
4. κ is **C**-definable;
5. if κ is not bottom, then $\hat{\jmath}\,\kappa \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1$, the procedure mapping the continuation procedure corresponding to κ to 1, is **D**-definable;
6. if κ and γ are inconsistent, then $[\hat{\jmath}\,\kappa \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1] \sqcup [\hat{\jmath}\,\gamma \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 0]$, the procedure mapping the continuation procedure for κ to 1 and the continuation procedure for γ to 0, is **D**-definable.

Proof

1. Just e. e can be either bottom, a number, or a procedure. If e is bottom, the defining term is Ω. If it is a number n, the defining term is $\ulcorner n \urcorner$.

   If e is a procedure, let $E = \{\ldots, p_r, \ldots\}$ be a minimal representation for e, where the p_r are precompact. Each p_r is of the form $a_{r,0} \Rightarrow \kappa_{r,0} \Rightarrow a_{r,1} \Rightarrow \kappa_{r,1} \Rightarrow \ldots a_{r,k_r} \Rightarrow \kappa_{r,k_r} \Rightarrow b_r$, where b_r is an observable. There are two cases.

   (a) Find, if possible, the smallest m such that E has two elements p_i and p_j whose $a_{i,m}$ and $a_{j,m}$, or $\kappa_{i,m}$ and $\kappa_{j,m}$ are inconsistent.

i. Let the inconsistent elements be $a_{i,m}$ and $a_{j,m}$. Then $(a_{i,m} \Rightarrow [\perp \Rightarrow \perp] \Rightarrow 1)$ and $(a_{j,m} \Rightarrow [\perp \Rightarrow \perp] \Rightarrow 0)$ are consistent, so $[(a_{i,m} \Rightarrow [\perp \Rightarrow \perp] \Rightarrow 1) \sqcup (a_{j,m} \Rightarrow [\perp \Rightarrow \perp] \Rightarrow 0)]$ exists, and by induction, is **D**-definable. Let the defining term be *Amij?*.

Let $E'_i = E \setminus \{p_i\}$ and $E'_j = E \setminus \{p_j\}$. Again, by induction, E'_i and E'_j are **D**-definable, by *Mi* and *Mj*, say.

Then e itself is **D**-definable by:

```
(lambda (x0)
  (call/cc'
    (lambda (c0)
      ...
      (lambda (xm)
        (call/cc'
          (lambda (cm)
            (abort
              (let*
                ([y1 (% (c0 ((if (Amij? xm)
                                Mj Mi)
                             x0)))]
                 [y2 (% (c1 (y1 x1)))]
                 ...
                 [r (% (cm (ym xm)))])
                r))))))))
```

ii. Let the inconsistent elements be $\kappa_{i,m}$ and $\kappa_{j,m}$. Then, by induction $([\hat{\jmath}(\kappa_{i,m}) \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1] \sqcup [\hat{\jmath}(\kappa_{j,m} \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 0])$ is **D**-definable. Let the defining term be *Kmij?*.

Let $E'_i = E \setminus \{p_i\}$ and $E'_j = E \setminus \{p_j\}$. E'_i and E'_j are respectively **D**-definable by *Mi* and *Mj*.

Then e is **D**-definable by the same term as in the previous case, with (*Kmij? cm*) substituted for (*Amij? xm*).

(b) Assume E does *not* have two elements such as p_i and p_j in the previous case. I.e., all the $a_{r,m}$ and $\kappa_{r,m}$ (r varies) are consistent. First, this implies that all the elements p_i of E are of the same length (say $2k+1$). Second, all the b_i must be the identical observable (say b). I.e., each p_i in E can be written as $a_{i,0} \Rightarrow \kappa_{i,0} \Rightarrow \ldots a_{i,k} \Rightarrow \kappa_{i,k} \Rightarrow b$.

By induction, each $a_{i,j} \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1$ is **D**-definable by $Aij?$, each $\hat{\jmath}(\kappa_{i,j}) \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1$ by $Kij?$, and b by B. Assuming that E consists of $\{p_0, \ldots, p_q\}$, e is **D**-defined by:

```
(lambda (x0)
  (call/cc'
    (lambda (c0)
      ...
      (lambda (xm)
        (call/cc'
          (lambda (cm)
            (pif (and ... (A0j? xj) (C0j? cj) ...)
                 B
                 (...
                  (pif (and ... (Aqj? xj) (Cqj? cj) ...)
                       B
                       Ω))))))
      ...)))
```

2. $e \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1$, where e is nonbottom. e is either a number or a procedure.

   (a) (e is a number n.) Define the predicates *n?* as in Chapter 3. Then $n \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1$ is **D**-defined by:

   (**lambda** (x)
     (*call/cc′*
       (**lambda** (c)
         (**if** (**and** (*number?* x) (*n?* x)) 1 Ω))))

   (b) (e is a procedure.) Let $E = \{\ldots, p_i, \ldots\}$ be a minimal representation of e, where each $p_i = a_i \Rightarrow \kappa_i \Rightarrow d_i$ is precompact. By induction, Ai defines a_i, $Ci[\,]$ **C**-defines κ_i, and *Di?* defines $d_i \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1$. Then $e \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1$ is defined by:

   (**lambda** (p)
     (**if** (**and** (*procedure?* p)
           $\ldots$ (*Di?* (% $Ci[(p\ Ai)]$)) $\ldots$)
       (**abort** 1) Ω))

3. $[e \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1] \sqcup [f \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 0]$, where e and f are inconsistent. There are three cases: Both e and f are numbers; only one of them is a number; both are procedures.

   (a) (e and f are numbers m and n respectively.) Clearly $m \neq n$. Without loss of generality, let $m < n$. Then the defining term is:

```
(lambda (x)
  (if (number? x)
     (if (m? x) (abort 1)
       (if (n? x) (abort 0)
         Ω))
     Ω))
```

(b) (Only one of e and f, say e, is a number, say m.) Let $F?$ define $f \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1$. Then, the defining term is:

```
(lambda (x)
  (if (and (number? x) (m? x)) (abort 1)
    (if (and (procedure? x) (F? x)) (abort 0)
      Ω)))
```

(c) (Both e and f are procedures.) Then, there is a $p = a_p \Rightarrow \kappa_p \Rightarrow d_p$ in E and a $q = a_q \Rightarrow \kappa_q \Rightarrow d_q$ in F such that p and q are inconsistent. This implies that a_p and a_q are consistent, as also κ_p and κ_q, but d_p and d_q are inconsistent.

Let R **D**-define $a_p \sqcup a_q$ and let $C[\ ]$ **C**-define $\kappa_p \sqcup \kappa_q$. Let $PQ?$ define $[d_p \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1] \sqcup [d_q \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 0]$. Let $E?$ and $F?$ respectively define $e \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1$ and $f \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1$. Then the defining term is:

```
(lambda (p)
  (if (not (procedure? p)) Ω
    (if (PQ? (p R))
       (E? p) (abort (not (% (F? p)))))))
```

4. Just κ. κ is either bottom or a function. If κ is bottom, the defining context is (**begin** Ω [ ]).

   If κ is a function, let $K = \{\ldots, q_r, \ldots\}$ be a minimal representation for κ, where the q_r are precompact. Each $q_r = a_{r,0} \Rightarrow d_r$, where d_r is a precompact element in $\mathbf{D}$ of the form $a_{r,1} \Rightarrow \kappa_{r,1} \ldots a_{r,k_r} \Rightarrow \kappa_{r,k_r} \Rightarrow b_r$, with b_r an observable. (There is no $\kappa_{r,0}$.) As in case 1(c), there are two possibilities.

   (a) Find, if possible, the smallest m such that K has two elements q_i and q_j whose $a_{i,m}$ and $a_{j,m}$, or $\kappa_{i,m}$ and $\kappa_{j,m}$ are inconsistent.

      i. Let the inconsistent elements be $a_{i,0}$ and $a_{j,0}$. Then $(a_{i,0} \Rightarrow [\perp \Rightarrow \perp] \Rightarrow 1) \sqcup (a_{j,0} \Rightarrow [\perp \Rightarrow \perp] \Rightarrow 0)$ is $\mathbf{D}$-definable, by *A0ij?*, say. Let $K_i' = K \setminus \{q_i\}$ be $\mathbf{C}$-definable by $Ci[\ ]$, and $K_j' = K \setminus \{q_j\}$ be $\mathbf{C}$-definable by $Cj[\ ]$. Then κ itself is $\mathbf{C}$-defined by:

      (**let** ([v [ ]])
        (**abort** (**pif** (*A0ij?* v) $Cj[v]$ $Ci[v]$)))

      ii. Let the inconsistent elements be $a_{i,m}$ and $a_{j,m}$ where $m \neq 0$. Then, as above, $[a_{i,m} \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1] \sqcup [a_{j,m} \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 0]$ is $\mathbf{D}$-definable, by *Amij?*, say.

      Let $K_i' = K \setminus \{q_i\}$ be $\mathbf{C}$-definable by $Ci[\ ]$, and $K_j' = K \setminus \{q_j\}$ be $\mathbf{C}$-definable by $Cj[\ ]$. Then κ itself is $\mathbf{C}$-defined by:

```
(let ([v [ ]])
  (abort
    (lambda (x1)
      (call/cc'
        (lambda (c1)
          ...
           (lambda (xm)
             (call/cc'
               (lambda (cm)
                 (abort
                   (let*
                     ([y1 (% (if (Amij? xm)
                                 Cj[v]
                                 Ci[v]))]
                      [y2 (% (c1 (y1 x1)))]
                      ...
                      [r (% (cm (ym xm)))])
                     r))))))))))
```

iii. Let the inconsistent elements be $\kappa_{i,m}$ and $\kappa_{j,m}$. Then, $[\hat{\jmath}(\kappa_{i,m}) \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1] \sqcup [\hat{\jmath}(\kappa_{j,m}) \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 0]$ is **D**-definable, by *Kmij?*, say. With the other defining terms as in the 4.b.i.B, κ is **C**-definable by the same term as above, with (*Kmij? cm*) substituted for (*Amij? xm*).

(b) Assume K does *not* have two elements q_i and q_j as above. I.e., all the $a_{r,m}$ and $\kappa_{r,m}$ (r varies) are consistent. This implies that all the q_r are of the same length (k say) and all the b_r are the same observable, b say.

Let *Aij?* define $a_{i,j} \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1$ and *Kij?* define $\hat{\jmath}(\kappa_{i,j}) \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1$. Assuming K consists of $\{q_0, \ldots, q_s\}$, κ is **C**-defined by:

```
(let ([v []])
  (abort
    (lambda (x1)
      (call/cc'
        (lambda (c1)
          ...
          (pif (and ... (A0j? xj) (C0j? cj) ...)
               B
               (...
                (pif (and ... (Asj? xj) (Csj? cj) ...)
                     B
                     Ω))))))))
```

5. $\hat{\jmath}\,\kappa \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1$, where κ is nonbottom.

   Let $\{\ldots, a_i \Rightarrow b_i, \ldots\}$ be a minimal representation for κ. Then $\{\ldots, a_i \Rightarrow (\perp \Rightarrow \perp) \Rightarrow b_i, \ldots\}$ is a minimal representation for $\hat{\jmath}\,\kappa$. $\hat{\jmath}\,\kappa \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1$ can then be defined using the same method as in 2.b.

6. $[\hat{\jmath}\,\kappa \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 1] \sqcup [\hat{\jmath}\,\gamma \Rightarrow (\perp \Rightarrow \perp) \Rightarrow 0]$, where κ and γ are inconsistent.

   Let $\{\ldots, a_i \Rightarrow b_i, \ldots\}$ be a minimal representation for κ and $\{\ldots, p_i \Rightarrow q_i, \ldots\}$ for γ. Then $\{\ldots, a_i \Rightarrow (\perp \Rightarrow \perp) \Rightarrow b_i, \ldots\}$ and $\{\ldots, p_i \Rightarrow (\perp \Rightarrow \perp) \Rightarrow q_i, \ldots\}$ minimally represent $\hat{\jmath}\,\kappa$ and $\hat{\jmath}\,\gamma$ respectively. The overall defining term is obtained using the same method as in 3(c). ∎

The proof of full abstraction of the cps model for PS+*call/cc'*+**pif**+% follows from the above lemma.

Theorem 4.1 The cps model is fully abstract for PS+*call/cc′*+**pif**+%.

Proof We need to show that if two closed terms M and N have different denotations, then they are observationally inequivalent. Let $M \not\equiv N$. I.e., $\mathcal{C}'[\![M]\!] = m$ and $\mathcal{C}'[\![N]\!] = n$ are different. We need to construct a program context $C[\ ]$ that distinguishes M and N.

1. The values m and n may be bottom, observable or procedural. Consider first the case where m and n belong to different categories among these three.

   (a) Distinguishing bottom from the rest. Let $m = \perp$ and let n be something else. Then the empty context distinguishes M and N.

   (b) Distinguishing numbers from procedures. Let m be a number and n a procedure. Then the context (*number*? [ ]) distinguishes the terms.

2. We next distinguish between values in the same category.

   (a) Distinguishing between numbers. A context consisting of simple arithmetic operations distinguishes the terms.

   (b) Distinguishing between procedures. Since m and n are differing procedures, there must be a precompact element in one—say m—that is not in the other. Let this element be $p = a_0 \Rightarrow c_0 \Rightarrow \ldots a_k \Rightarrow c_k \Rightarrow b$, where b is an observable. By Lemma 4.2, the Ai define a_i, $Ci[\ ]$ define c_i and B defines b. Then the context

$$
\begin{array}{l}
(Ck[(\ldots \\
\qquad (\%\ C1[((\%\ C0[([\ ]\ A0)])\ A1)]) \\
\qquad \ldots \\
\qquad Ak)])
\end{array}
$$

distinguishes the terms. ∎

Chapter 5

Modeling PS+control directly

The basic direct model of Chapter 3 can be extended to accommodate enhancements of PS that include control operators [19, 48]. The extensions consist in (1) adding to the domain *extension values* that account for the meanings of the new terms in the language; (2) adding clauses to the meaning function for the new terms; (3) modifying the *apply* semantic function to reflect the effect of the extension values; and (4) providing a semantic function *strip* that decomposes extension values.

The template for the domain equation remains the same across the several kinds of extended models, including the unextended basic model. Only one specification, namely that of the subdomain for the extension values, need be changed. This does not mean that the other, non-extension, values remain identical to their basic-model counterparts. Since the domain equation is recursive, the composition of the extension subdomain affects the values in the other subdomains. Thus, for example, the values in the procedure subdomain now include procedures that may yield extension values. However, the extended domain remains similar enough to the original domain that values can be projected and injected across the two with fair ease and clarity [19].

Looking closer at the domain equation, we note that the domain for the plain PS language satisfies:

$$\mathbf{D} = \overbrace{\mathbf{O}}^{\text{integers}} \oplus \overbrace{(\mathbf{D} \to_s \mathbf{D})_\perp}^{\text{procedures}} .$$

This can also be written as:

$$\mathbf{D}' \;=\; \overbrace{\mathbf{O}}^{\text{integers}} \oplus \overbrace{(\mathbf{D}' \rightarrow_s \mathbf{D})_\perp}^{\text{procedures}}$$

$$\mathbf{D} \;=\; \mathbf{D}'$$

where $\mathbf{D}'$ is merely a synonym for $\mathbf{D}$.

An *extended* domain $\mathbf{D}_e$ introduces an additional subdomain to account for the denotations introduced by the language extension. Thus, any standard extension of the basic direct model can be formulated as the following equation, which differs from the above only in adding an extension subdomain built—using some domain construction Γ—to suit the particular language extension:

$$\mathbf{D}'_e \;=\; \overbrace{\mathbf{O}}^{\text{integers}} \oplus \overbrace{(\mathbf{D}'_e \rightarrow_s \mathbf{D}_e)_\perp}^{\text{procedures}}$$

$$\mathbf{D}_e \;=\; \underbrace{\mathbf{D}'_e}_{\text{regular values}} \boxed{\oplus \;\; \underbrace{\Gamma(\mathbf{D}'_e, \mathbf{D}_e)}_{\text{extension values}}}$$

(As usual, we use the tags inL and inR to refer to the elements in the *l*eft and *r*ight subdomains respectively of a disjoint union.) The new elements have the following characteristics:

1. Extension values are typically regular values embellished with a different tag. Thus, $\Gamma(\mathbf{D}'_e, \mathbf{D}_e)$ is often simply $\mathbf{D}'_e$.

2. Applications in an extended language are different from the those in the base language. This is reflected in the model by modifying *apply* to recognize extension values.

3. Extension values are volatile—they eventually dissolve into something built out of their constituent regular values. The semantic function *strip* is used to de-

compose extension values. An instance of the use of *strip* is in arriving at a program's denotation, which must be a regular value.

The semantic function $\mathcal{A}$ of the basic model is enhanced to $\mathcal{A}_e$ by adding clauses that provide meanings for the new terms in the language. The clauses for the original terms remain the same. The environments map variables to values that are *neither bottom nor extension values.*

The meaning function $\mathcal{A}'_e$ for programs is defined as:

$$\mathcal{A}'_e : PS_e \rightarrow \mathbf{D}'_e$$
$$\mathcal{A}'_e[\![M]\!] = strip(\mathcal{A}_e[\![M]\!]\bot)$$

5.1 Modeling PS+abort

Extending PS with the control operator **abort** illustrates the technique described above. The extension values that need to be added are called *abort values*, and they are merely tagged versions of the regular values:

$$\mathbf{D}'_a = \overbrace{\mathbf{O}}^{\text{integers}} \oplus \overbrace{(\mathbf{D}'_a \rightarrow_s \mathbf{D}_a)_\bot}^{\text{procedures}}$$

$$\mathbf{D}_a = \underbrace{\mathbf{D}'_a}_{\text{regular values}} \oplus \underbrace{\mathbf{D}'_a}_{\text{abort values}}$$

For readability, the tag inA is used instead of inR to refer to the elements in the subdomain for Abort-values. The semantic function $\mathcal{A}_a$ maps terms in PS+**abort** through environments to values in $\mathbf{D}_a$, and is similar to the function $\mathcal{A}$ (of PS) for the non-**abort** terms. Indeed, $\mathbf{D}_a$ could well be viewed as a model for PS (without **abort**). However it is not fully abstract for this language.

Claim 5.1 The extended direct model $\mathbf{D}_a$ is not fully abstract for PS.

Proof Consider the terms

$$\begin{aligned} P_0 &\equiv (\textbf{lambda}\ (f)\ \Omega) \\ P_1 &\equiv (\textbf{lambda}\ (f)\ ((f\ 1)\ \Omega)) \end{aligned}$$

from Claim 3.1. Using a similar proof, these terms are observationally equivalent in PS with the meanings provided by $\mathbf{D}_a$. However, the model distinguishes them using

$$a = \mathsf{inL}(\mathsf{inP}(\lambda v.\, \mathsf{inA}(v)))$$

For $apply(\mathcal{A}_a[\![P_u]\!]\bot, a)$ is $\bot$ and $\mathsf{inA}(1)$ for $u = 0$ and 1 respectively. ∎

Adding **abort** to PS dispels this counterexample, for the terms P_1 and P_2 are no longer observationally equivalent. Indeed, the context

$$([\,]\ (\textbf{lambda}\ (v)\ (\textbf{abort}\ v)))$$

distinguishes the two.

The additional semantic clause for **abort** is:

$$\mathcal{A}_a[\![(\textbf{abort}\ M)]\!]\rho = \mathsf{inA}(strip(\mathcal{A}_a[\![M]\!]\rho))$$

The new version of *apply* depicts how abort values ignore their surrounding context. Given an application, the first of the following clauses that matches is invoked:

$$apply : \mathbf{D}_a \times \mathbf{D}_a \to \mathbf{D}_a$$

$$\begin{aligned} apply(\mathsf{inA}(a), b) &= \mathsf{inA}(a) && \text{procedure position aborts} \\ apply(a, \mathsf{inA}(b)) &= \mathsf{inA}(b) && \text{argument position aborts} \\ apply(\mathsf{inL}(\mathsf{inP}(a)), \mathsf{inL}(b)) &= ab && \text{plain procedure application} \\ apply(a, b) &= \bot && a \text{ is observable or } \bot \end{aligned}$$

The order given here chooses *left-to-right* evaluation. Interchanging the first two clauses gives *right-to-left* evaluation of applications.

The function *strip* is used to remove the tag associated with **abort**-values to yield the result of a program. To get the denotation of a program P, find the meaning of the *term* P, and use *strip* to extract the regular value:

$$\mathcal{A}'_a[\![M]\!] = \mathit{strip}(\mathcal{A}_a[\![M]\!]\bot)$$

where *strip* is given by:

$$\mathit{strip} : \mathbf{D}_a \rightarrow \mathbf{D}'_a$$

$$\begin{aligned} \mathit{strip}(\mathsf{inA}(a)) &= a \\ \mathit{strip}(a) &= a \end{aligned}$$

Like $\mathbf{D}$ (Chapter 3), this extended domain $\mathbf{D}_a$ also supports parallel disjunction. PS+**abort** lacks a facility for doing this, and the same counterexample as in Claim 3.2 bears witness to the failure of full abstraction. Thus, if PS+**abort** is to be fully abstract, it must include a parallel-**if**. However, the construct differs slightly from that for plain PS, since it must accommodate control action in the branches. The semantics of *pif** is:

$$\mathcal{A}_a[\![\mathit{pif}*]\!]\rho = \underline{\mathit{pif}*}$$

where $\underline{\mathit{pif}*}$ is the function:

$$\lambda b, t, e. \begin{cases} \mathit{thaw}\ e & \text{if } \mathit{thaw}\ b \text{ is } 0 \\ \mathit{thaw}\ t & \text{if it is neither 0 not } \bot \\ n & \text{if } \mathit{thaw}\ t = \mathit{thaw}\ e = n\text{, a number} \\ \multicolumn{2}{l}{\mathsf{inL}(\mathsf{inP}(\lambda v.\underline{\mathit{pif}*}\ b\ (\mathit{freeze}((\mathit{thaw}\ t)v))\ (\mathit{freeze}((\mathit{thaw}\ e)v)))} \\ & \text{if } \mathit{thaw}\ t \text{ and } \mathit{thaw}\ e \text{ are both procedures} \\ \multicolumn{2}{l}{\mathsf{inA}(\mathit{strip}(\underline{\mathit{pif}*}\ b\ (\mathit{freeze}(p))\ (\mathit{freeze}(q))))} \\ & \text{if } \mathit{thaw}\ t = \mathsf{inA}(p) \text{ and } \mathit{thaw}\ e = \mathsf{inA}(q) \end{cases}$$

where

$$\begin{aligned} \mathit{thaw}\ \theta &= \theta\ (\mathsf{inL}(\mathsf{inO}(0))) \\ \mathit{freeze}\ v &= \mathsf{inL}(\mathsf{inP}(\lambda\delta.v)) \end{aligned}$$

The first four subclauses in the $\underline{\mathit{pif*}}$ equation are identical to the one for PS and **D**. In the final, new subclause where both branches are **abort**s, the entire *pif**-call is converted to an **abort** after replacing both branches of the *pif** with procedures. This, as per the previous rule, postpones the decision of whether the branches are equal to the site that receives the **abort**.

5.2 Abort and prompt

Parallel disjunction is not the only construct that the language PS+**abort** needs in order to make the model fully abstract. This is because models for control usually can also *delimit* control [47]. In the case of the extended direct model for PS+**abort**, the function $\lambda\theta.\mathit{strip}(\mathit{thaw}(\theta))$ provides this control-delimiting capability. For example, this function can distinguish the denotations of the terms M_u $(u = 0, 1)$ defined as follows:

$$\begin{aligned} M_u \stackrel{\text{df}}{=} (&\textbf{lambda}\ (z) \\ &\quad (\mathit{equal?}\ \ (z\ (\textbf{lambda}\ (d)\ (\textbf{abort}\ 1))) \\ &\qquad\qquad\quad (z\ (\textbf{lambda}\ (d)\ (\textbf{abort}\ \ulcorner u \urcorner))))) \end{aligned}$$

However, the *language* PS+**abort**, even enhanced with **pif**, cannot distinguish these terms—a distinguishing context would have to provide an argument z that invokes the **abort**s *and* performs the *equal?* test. But, in this case, the invocation of the subterm (**abort** 1) immediately exits the program with the result 1. Thus, the other **abort**-call and the *equal?* test have no opportunity to occur, and the two terms are indistinguishable.

An **abort**-delimiting procedure, on the other hand, can capture the **abort**ed values before the procedure z exits. The procedure *equal?* can thus examine both the **abort**-values, yielding differing results for M_0 and M_1.

Such an **abort**-delimiter is the *prompt* operator % [16]. Its semantics is given by:

$$\mathcal{A}_a[\![\%\ M]\!]\rho = \mathsf{inL}(strip(\mathcal{A}_a[\![M]\!]\rho))$$

Using the prompt, the distinguishing context for M_0 and M_1 is:

$$([\ ]\ (\textbf{lambda}\ (th)\ (\%\ (th\ 0))))$$

5.3 Full abstraction for PS+abort+pif+%

To prove that the model $\mathbf{D}_a$ is fully abstract for PS+**abort**+**pif**+%, we follow the same outline as in the proof for PS+**pif**. First we prove a lemma that compact elements are definable, and then use this result to show that denotational and observational equivalences match.

As in Lemma 3.3, which defines compact elements for plain PS, the definability lemma for PS+**abort** also defines procedures mapping the compact elements to boolean values. However, since some of the compact elements are abort values that are too volatile to serve as procedure arguments, the boolean procedures map the *thunk* or *frozen* form of the element, rather than the element itself. The thunk form of a value v is $0 \Rightarrow v$, i.e., a procedure that can be "discharged" or "thawed" to yield v, by applying it to the argument 0.

Lemma 5.1 For all compact elements e and f in $\mathbf{D}_a$:

1. e is definable;

2. if e is non-bottom,[11] then $(0 \Rightarrow e) \Rightarrow 1$, the procedure mapping the thunk form of e to 1, is definable; and

3. if e and f are inconsistent, then $[(0 \Rightarrow e) \Rightarrow 1] \sqcup [(0 \Rightarrow f) \Rightarrow 0]$, the procedure mapping the thunk form e to 1 and the thunk form of f to 0, is definable.

Proof The proof is an induction on the *size* of the compact elements, as in the proof of Lemma 3.3.[12]

1. Just e. e can be either bottom, number, procedure or abort value.

   (a) (e is bottom.) The defining term is Ω.

   (b) (e is a number n.) The defining term is $\ulcorner n \urcorner$.

   (c) (e is a procedure.) Let $E = \{\ldots p_i \ldots\}$ be *a* minimal representation for e, where the p_i are precompact. Each p_i is of the form $a_{i,0} \Rightarrow \ldots a_{i,k_i} \Rightarrow b_i$, where each $a_{i,j}$ is an observable or compact procedure, and b_i is an observable or precompact abort value. There are two cases.

      i. First, assume that for each m, the various $a_{r,m}$ are consistent. As in Lemma 3.3, all the p_i are of the same length, and all the b_i are consistent. *Unlike* in that lemma, however, the b_i need not be identical—bottom or observable. They can be consistent but different precompact abort values.

      Nevertheless, using the new **pif**, which allows abort values in its arms, the defining term remains, as in Lemma 3.3:

[11]This predicate is certainly definable for $e = \perp$, but we do not need this result.

[12]Each tag also contributes a size unit.

$$
\begin{array}{l}
(\textbf{lambda}\ (x_0 \ldots\ x_k) \\
\quad (\textbf{pif}\ (\textbf{and} \ldots\ (An?\ (\textbf{lambda}\ (d)\ x_n)) \ldots) \\
\qquad Bi\ (Mi \ldots\ x_n \ldots)))
\end{array}
$$

where, given $p_i \equiv a_{i,0} \Rightarrow \ldots a_{i,k} \Rightarrow b_i$ is one of E's precompact elements, $An?$ defines $(0 \Rightarrow a_{i,n}) \Rightarrow 1$ for each n; Bi defines b_i; and Mi defines $E'_i = E \setminus \{p_i\}$.

ii. Find, if possible, the smallest m such that E has two elements p_i and p_j where $a_{i,m}$ and $a_{j,m}$ are inconsistent. As in Lemma 3.3, this implies that e is a (curried) procedure of at least $m+1$ arguments. As in that lemma, the defining term for e is:

$$
\begin{array}{l}
(\textbf{lambda}\ (x_0 \ldots\ x_m) \\
\quad (\textbf{pif}\ (Aij?\ (\textbf{lambda}\ (d)\ x_m)) \\
\qquad (Mj\ x_0 \ldots\ x_m)\ (Mi\ x_0 \ldots\ x_m)))
\end{array}
$$

where $Aij?$ defines $[(0 \Rightarrow a_{i,m}) \Rightarrow 1] \sqcup [(0 \Rightarrow a_{j,m}) \Rightarrow 0]$; Mi defines $E'_i = E \setminus \{p_i\}$; and Mj defines $E'_j = E \setminus \{p_j\}$.

(d) (e is an abort value.) Let $e = \mathsf{inA}(g)$, where g is clearly a compact, *non*-abort value. Using the previous cases, g is definable by M say. Then e is defined by:

$$(\textbf{abort}\ M)$$

2. $(0 \Rightarrow e) \Rightarrow 1$, procedure mapping the thunk form of e to 1, where e is non-bottom, i.e., a number, procedure or abort value.

   For this, we need a predicate in the language that tests if its argument thunk aborts:

```
aborting-thunk? ≝
  (lambda (th)
    (let ([x (% (begin (th 0) 0))])
      (let ([y (% (begin (th 0) 1))])
        (not (different-numbers? x y)))))
```

If th aborts, then x and y must be the same value. If th does not abort, then $x = 0$ and $y = 1$. The predicate *different-numbers?* returns true if its arguments are different numbers:

```
different-numbers? ≝
  (lambda (m n)
    (and (number? m) (number? n) (not (equal? m n))))
```

(a) (e is a number n.) Define the predicates *n?* as in Chapter 3. Then, $(0 \Rightarrow n) \Rightarrow 1$ is defined by:

```
(lambda (th)
  (if (aborting-thunk? th) Ω
    (let ([x (th 0)])
      (if (and (number? x) (n? x)) 1 Ω))))
```

(b) (e is a procedure.) Let $E = \{\ldots, a_i \Rightarrow b_i, \ldots\}$. Let Ai define a_i and *Bi?* define $(0 \Rightarrow b_i) \Rightarrow 1$. Then $(0 \Rightarrow e) \Rightarrow 1$ is defined by:

```
(lambda (th)
  (if (aborting-thunk? th) Ω
    (let ([x (th 0)])
      (if (and (procedure? x)
                 ... (Bi? (lambda (d) (x Ai))) ...)
        1 Ω))))
```

(c) (e is an abort value.) Let $e = \mathsf{inA}(g)$, where g is clearly compact. From the other cases, $(0 \Rightarrow g) \Rightarrow 1$ is definable, by *G?* say. Then $(0 \Rightarrow e) \Rightarrow 1$ is defined by:

```
(lambda (th)
  (if (not (aborting-thunk? th)) Ω
    (let ([x (% (th 0))])
      (if (G? (lambda (d) x)) 1 Ω))))
```

3. $[(0 \Rightarrow e) \Rightarrow 1] \sqcup [(0 \Rightarrow f) \Rightarrow 0]$, procedure mapping the thunk forms of e, f to $1, 0$ respectively, where e and f are inconsistent. There are four cases: Both e and f are numbers; only one of them is a number; both are procedures;

(a) (Both are numbers.) Let e and f be the numbers m and n respectively. Clearly, $m \neq n$: let $m < n$. Then, the defining term is:

```
(lambda (th)
  (if (aborting-thunk? th) Ω
    (let ([x (th 0)])
      (if (not (number? x)) Ω
        (if (m? x) 1
          (if (n? x) 0 Ω))))))
```

(b) (Both are procedures.) Let E, F be the minimal representations of e, f respectively. Since e, f are inconsistent, there must be a $p = p_0 \Rightarrow p_1$ in E and a $q = q_0 \Rightarrow q_1$ in F such that p, q are inconsistent. I.e., p_0, q_0 are consistent, but p_1, q_1 are not.

Let *P0-&-Q0* define $p_0 \sqcup q_0$; *P1-V-Q1?* define $[(0 \Rightarrow p_1) \Rightarrow 1] \sqcup [(0 \Rightarrow q_1) \Rightarrow 0]$; *E?* define $(0 \Rightarrow e) \Rightarrow 1$; and *F?* define $(0 \Rightarrow f) \Rightarrow 1$. Then the following term defines $[(0 \Rightarrow e) \Rightarrow 1] \sqcup [(0 \Rightarrow f) \Rightarrow 0]$:

$$
\begin{array}{l}
(\textbf{lambda}\ (th) \\
\quad (\textbf{if}\ (\textit{aborting-thunk?}\ \ th)\ \Omega \\
\quad\quad (\textbf{let}\ ([x\ (th\ 0)]) \\
\quad\quad\quad (\textbf{if}\ (\textit{not}\ (\textit{procedure?}\ x))\ \Omega \\
\quad\quad\quad\quad (\textbf{if}\ (\textit{P1-V-Q1?}\ (\textbf{lambda}\ (d)\ (x\ \textit{P0-\&-Q0}))) \\
\quad\quad\quad\quad\quad (\textit{E?}\ th)\ (\textit{not}\ (\textit{F?}\ th)))))))
\end{array}
$$

(c) (Both are abort values.) Then $e = \mathsf{inA}(u), f = \mathsf{inA}(v)$, where g, h are compact, non-abort values. Note that g, h are also inconsistent. Thus, $[(0 \Rightarrow g) \Rightarrow 1] \sqcup [(0 \Rightarrow h) \Rightarrow 0]$ is definable, by *G-V-H?*, say. Then $[(0 \Rightarrow e) \Rightarrow 1] \sqcup [(0 \Rightarrow f) \Rightarrow 0]$ is defined by:

$$
\begin{array}{l}
(\textbf{lambda}\ (th) \\
\quad (\textbf{if}\ (\textit{not}\ (\textit{aborting-thunk?}\ \ th))\ \Omega \\
\quad\quad (\textbf{let}\ ([x\ (\%\ (th\ 0))]) \\
\quad\quad\quad (\textit{G-V-H?}\ (\textbf{lambda}\ (d)\ x)))))
\end{array}
$$

(d) (e, f belong to different subdomains (number, procedure, abort value).) The corresponding defining terms use the predicates *aborting-thunk?* to distinguish abort values from the rest, and then *number?* to distinguish numbers from procedures. After this check, they invoke the terms for either $(0 \Rightarrow e) \Rightarrow 1$ or $(0 \Rightarrow f) \Rightarrow 1$. These, as shown above, are definable, say by *E?* and *F?* respectively.

i. (One is an abort value, the other is a number or procedure.) Let e be the abort value. The defining term is:

$$
\begin{array}{l}
(\textbf{lambda}\ (th) \\
\quad (\textbf{if}\ (\textit{aborting-thunk?}\ \ th) \\
\quad\quad (\textit{E?}\ th)\ (\textit{not}\ (\textit{F?}\ th))))
\end{array}
$$

ii. (One is a number, the other a procedure.) Let e be the number. The defining term is:

$$\begin{array}{l}(\textbf{lambda}\ (th)\\ \quad (\textbf{if}\ (\textit{aborting-thunk?}\ \ th)\ \Omega\\ \quad\quad (\textbf{let}\ ([x\ (th\ 0)])\\ \quad\quad\quad (\textbf{if}\ (\textit{number?}\ \ x)\ (E?\ th)\ (\textit{not}\ (F?\ th)))))) \ \blacksquare\end{array}$$

Theorem 5.1 The extended direct model $\mathbf{D}_a$ is fully abstract for PS+**abort**+**pif**+%.

Proof The proof of full abstraction follows from the above lemma, using a strategy similar to that for PS+**pif** (Lemma 3.2). We need to show that if the closed terms M and N have different meanings, then they are observationally distinguishable.

Let $\mathcal{A}'_a[\![M]\!] = m$ and $\mathcal{A}'_a[\![N]\!] = n$ with $m \neq n$. We now need to show a context $C[\,]$ such that $C[M]$ and $C[N]$ differ in their termination behavior.

1. The values m and n can be bottom, observable, procedure or abort value. Consider first the case where m and n belong to different categories among these four.

   (a) Distinguishing bottom from observables, procedures and abort values. Let $m = \perp$ and n anything else. Then the empty context distinguishes M and N.

   (b) Distinguishing abort values from observables and procedures. Let m be an abort value and n an observable or procedure. Then the context

   $$(\textit{aborting-thunk?}\ \ (\textbf{lambda}\ (d)\ [\,]))$$

   distinguishes M and N.

(c) Distinguishing observables from procedures. Let m be a number and n a procedure. Then the context

$$(\textit{number?}\ [\,])$$

can distinguish M, N.

2. We next must distinguish between different values in the same category.

(a) Distinguishing between numbers. A context using simple arithmetic operations distinguishes M and N.

(b) Distinguishing between procedures. Since m and n are different procedure values, there must be a precompact procedure element in one—say m—that is not in the other—say n. Let this element be $p = a_0 \Rightarrow \ldots a_k \Rightarrow b$. By Lemma 5.1, the a_i are definable by A_i, and b by B. Then, applying the terms M, N respectively to A_0, ..., A_k gives B in the case of M and something else for N, which can then be distinguished using the other cases.

(c) Distinguishing between abort values. Let $m = \mathsf{inA}(p)$ and $n = \mathsf{inA}(q)$. The values p and q are definable and can be distinguished using the other cases. Let this distinguishing context be $C[\,]$. Then M and N are distinguished by $C[(\%\ [\,])]$.

Thus if $M \not\equiv N$, we can find a program context showing that $M \not\approx N$. Hence, the model is fully abstract. ∎

5.4 Modeling PS + a higher-order control operator

Extended direct models can also be built for control operators more powerful than **abort**. This section will consider PS+*fcontrol*, where *fcontrol* is the higher-order control operator introduced in Chapter 2. The domain equation constructing the domain $\mathbf{D}_f$ for PS+*fcontrol* is:

$$\begin{array}{rcl}
\mathbf{D}'_f & = & \overbrace{\mathbf{O}}^{\text{integers}} \oplus \overbrace{\mathbf{P}_f}^{\text{procedures}} \\
\mathbf{D}_f & = & \underbrace{\mathbf{D}'_f}_{\text{regular values}} \oplus \underbrace{(\mathbf{D}'_f \otimes \mathbf{P}_f)}_{\textit{fcontrol}\text{-values}} \\
\mathbf{P}_f & = & (\mathbf{D}'_f \rightarrow_s \mathbf{D}_f)_\perp
\end{array}$$

For readability, the tag inF is used of inR to refer to the elements in the subdomain for the *fcontrol*-values. The semantic function $\mathcal{A}_f$ maps terms in PS+*fcontrol* through environments to values in $\mathbf{D}_f$, and is the same as the function $\mathcal{A}$ for PS and $\mathcal{A}_a$ for PS+**abort** for the terms not involving *fcontrol*.

The additional clause for *fcontrol* reads:

$$\mathcal{A}_f[\![(\textit{fcontrol}\ M)]\!]\rho = \mathsf{inF}(\mathcal{A}_f[\![M]\!]\rho, \lambda v.v)$$

The difference in the control strategy is encoded by the *apply* semantic function, whose clauses are:

$$\textit{apply} : \mathbf{D}_f \times \mathbf{D}_f \rightarrow \mathbf{D}_f$$

$$\begin{array}{rcl}
\textit{apply}(\mathsf{inF}(r,k),a) & = & \mathsf{inF}(r, \lambda v.k(\textit{apply}(v,a))) \\
\textit{apply}(a, \mathsf{inF}(r,k)) & = & \mathsf{inF}(r, \lambda v.k(\textit{apply}(a,v))) \\
\textit{apply}(\mathsf{inL}(\mathsf{inP}(a)),b) & = & ab \\
\textit{apply}(a,b) & = & \perp
\end{array}$$

Once again, program denotations can only be regular values and a *strip* function is used to convert *fcontrol*-values to regular ones. The program meaning function is:

$$\mathcal{A}'_f[\![M]\!] = \textit{strip}(\mathcal{A}[\![M]\!]\perp, \lambda v.v)$$

where *strip* uses its second argument to *discharge* an *fcontrol*-value:

$$strip : \mathbf{D}'_f \times \mathbf{P}_f \rightarrow_s \mathbf{D}'_f$$

$$\begin{aligned} strip(\mathsf{inF}(r,k),b) &= apply(br,k) \\ strip(a,b) &= a \end{aligned}$$

Like the direct models for PS and PS+**abort**, the model for PS+*fcontrol* is also capable of parallel disjunction. Thus, adding parallel-**if** to PS+*fcontrol* is a first step toward ensuring full abstraction. The semantics is identical to the one given above for PS+**abort**, i.e., when both branches represent control actions, the control action is applied directly to the **pif**-call:

$$\mathcal{A}_f[\![pif*]\!]\rho = \underline{pif*}$$

where $\underline{pif*}$ is the function:

$$\lambda b,t,e. \begin{cases} \ldots & \text{as in Section 5.1} \\ \mathsf{inF}(p,g) \text{ if} & thaw\ t = \mathsf{inF}(r,k) \\ & thaw\ e = \mathsf{inF}(q,j) \\ & p = \underline{pif*}\ b\ r\ q \\ & g' = \underline{pif*}\ b\ \mathsf{inL}(\mathsf{inP}(k))\ \mathsf{inL}(\mathsf{inP}(j)) \\ & \mathsf{inL}(\mathsf{inP}(g)) = g' \end{cases}$$

The model $\mathbf{D}_f$ is more powerful than the previous extended model $\mathbf{D}_a$. To be sure, $\mathbf{D}_f$ models PS+**abort**+**pif**+**%** with the meaning clauses:

$$\begin{aligned} \mathcal{A}_f[\![(\mathbf{abort}\ M)]\!]\rho &= \mathsf{inF}(\mathcal{A}_f[\![M]\!]\rho, \lambda v.0) \\ \mathcal{A}_f[\![(\%\ M)]\!]\rho &= \mathsf{inL}(strip'(\mathcal{A}_f[\![M]\!]\rho)) \end{aligned}$$

where *strip*$'$ ignores the second element of an inF pair. However, $\mathbf{D}_f$ contains too many capabilities to be fully abstract for the restricted language.

Claim 5.2 The extended direct model $\mathbf{D}_f$ is not fully abstract for PS+**abort**+**pif**+%.

Proof Consider the terms:

$$
\begin{aligned}
M_u \stackrel{\text{df}}{=} (&\textbf{lambda}\ (f)\\
&\quad(\textbf{begin}\\
&\quad\quad(f\ (\textbf{lambda}\ (d)\ 1))\\
&\quad\quad(f\ (\textbf{lambda}\ (d)\ \Omega))\\
&\quad\quad(f\ (\textbf{lambda}\ (d)\ \ulcorner u \urcorner))\\
&\quad\quad(f\ (\textbf{lambda}\ (d)\ \Omega))))
\end{aligned}
$$

for $u = 0$ and 1. Consider any program context $C[\ ]$, and assume that $C[M_0]$ converges. Then, by a version of Lemma 3.2, the meaning $\mathcal{A}_f[\![C[M_0]]\!]\bot$ is non-bottom. By the Activity Lemma (Section 2.9), either $C[M_1]$ converges, or for all procedural M:

$$C[M] \stackrel{*}{\triangleright} E[(M\ U)]$$

for some value U. In particular,

$$C[M_u] \stackrel{*}{\triangleright} E[(M_u\ U)]$$

for $u = 0$ and 1. By specializations of Lemmas 3.2 and 3.1, the convergence of $\mathcal{A}'_f[\![C[M_u]]\!]$ depends on the convergence of $\mathcal{A}'_f[\![(M_u\ U)]\!]\bot$, which is:

$$
\mathcal{A}'_f\left[\begin{array}{l}
(\textbf{begin}\ (f\ (\textbf{lambda}\ (d)\ 1))\\
\qquad\quad (f\ (\textbf{lambda}\ (d)\ \Omega))\\
\qquad\quad (f\ (\textbf{lambda}\ (d)\ \ulcorner u \urcorner))\\
\qquad\quad (f\ (\textbf{lambda}\ (d)\ \Omega)))
\end{array}\right]\bot[f/v]
$$

where v is $\mathcal{A}_f[\![U]\!]\bot$.

There are three cases: The procedure v either (1) ignores its argument; (2) aborts it; or (3) applies it. If the first and second cases, there is the programs behave identically for both values of u. Since $C[M_0]$ converges (i.e., has a meaning of non-bottom), so too must $C[M_1]$. In the third case, both programs diverge, in particular $C[M_0]$, contradicting the assumption. Thus we have shown that $C[M_1]$ converges whenever $C[M_0]$ does. By a similar reasoning, we can show that $C[M_0]$ converges whenever $C[M_1]$ does. Thus, the terms M_0 and M_1 are observationally equivalent. ∎

In other words, PS enhanced with first-order control constructs (and parallel disjunction) is not enough to ensure full abstraction of the model.

5.5 The new prompt

The control actions in PS+*fcontrol* also need a control delimiter, but in a substantially different form. The naive single-argument prompt simply mimics $\mathcal{A}'_f$ to get a program within a program, as in PS+**abort**:

$$\mathcal{A}[\![(\%\ M)]\!]\rho = \mathit{strip}(\mathcal{A}[\![M]\!]\rho, \lambda v.v)$$

However, this is not enough to distinguish expressions like

$$(\mathit{fcontrol}\ (\mathbf{lambda}\ (k)\ 3))$$

and

$$(\mathit{fcontrol}\ (\mathbf{lambda}\ (k)\ (\mathit{fcontrol}\ (\mathbf{lambda}\ (k)\ 3))))$$

The second term differs from the first in activating both *fcontrol*-expressions, if only to exit the program with the same final answer. However the denotations in $\mathbf{D}'_f$ are different, viz., $\mathsf{inF}(\mathsf{inP}(\lambda k.3), \lambda v.v)$ and $\mathsf{inF}(\mathsf{inP}(\lambda k.\,\mathsf{inF}(\mathsf{inP}(\lambda k.3))), \lambda v.v)$ respectively.

A better control delimiter for PS+*fcontrol* should therefore be able to keep track of the *different* *fcontrol*-applications that happen within its perimeter. This is done by having a two-argument prompt, reflecting the fact that the semantic function *strip* has two arguments. The second argument is called a *handler* (Chapter 2). The semantic clause for this prompt is:

$$\mathcal{A}_f[\![(\%\ M\ H)]\!]\rho = \mathit{strip}(\mathcal{A}_f[\![M]\!]\rho, h)$$

where

$$\mathcal{A}_f[\![H]\!]\rho = \mathsf{inL}(\mathsf{inP}(h))$$

Thus the prompt-with-a-handler captures $\mathbf{D}_f$'s *strip* operator exactly, just as PS+**abort**'s unary prompt captured *its* model's *strip*. The extended prompt distinguishes the two terms above using the handler

$$(Y\ (\textbf{lambda}\ (h)\ (\textbf{lambda}\ (r\ k)\ (+\ 1\ (\%\ (r\ k)\ h)))))$$

where Y is the fixpoint combinator used to get recursive functions:

$$\begin{array}{l} Y \stackrel{\text{df}}{=} (\textbf{lambda}\ (f) \\ \qquad\quad (\textbf{let}\ ([w\ (\textbf{lambda}\ (x)\ (f\ (\textbf{lambda}\ (z)\ ((x\ x)\ z))))]) \\ \qquad\qquad (w\ w))) \end{array}$$

The new language also distinguishes the terms M_0 and M_1 of Claim 5.2. The context that tells these terms apart is:

$$\begin{array}{l} (\%\ ([\ ]\ \mathit{fcontrol}) \\ \quad (\textbf{lambda}\ (r\ k) \\ \quad\quad (\%\ (k\ (r\ 0)) \\ \quad\quad\quad (\textbf{lambda}\ (r\ k) \\ \quad\quad\quad\quad (\%\ (k\ 0) \\ \quad\quad\quad\quad\quad (\textbf{lambda}\ (r\ k)\ (r\ 0))))))) \end{array}$$

The first *fcontrol*-application is resumed by the handler after discharging its thunk. The second *fcontrol*-application is also resumed, but its thunk is avoided. The third *fcontrol*-application is used to throw the discriminating value, $\ulcorner u \urcorner$, to its handler. The last *fcontrol*-application is avoided altogether. Thus the terms yield 0 and 1 respectively, for $u = 0, 1$, so they are *not* observationally equivalent.

5.6 Full abstraction for PS+*fcontrol*+pif+%

The proof that the model $\mathbf{D}_f$ is fully abstract for PS+*fcontrol*+**pif**+% follows the same path as for PS+**abort**+**pif**+%. First we prove a lemma that compact elements are definable, and then use this result to show that denotational and observational equivalences match.

The definability predicate for the compact elements of $\mathbf{D}_f$ is identical to the one for $\mathbf{D}_a$.

Lemma 5.2 For all compact elements e and f in $\mathbf{D}_f$:

1. e is definable;
2. if e is non-bottom, then $(0 \Rightarrow e) \Rightarrow 1$, the procedure mapping the thunk form of e to 1, is definable; and
3. if e and f are inconsistent, then $[(0 \Rightarrow e) \Rightarrow 1] \sqcup [(0 \Rightarrow f) \Rightarrow 0]$, the procedure mapping the thunk form of e to 1 and the thunk form of f to 0, is definable.

Proof The proof induces on the *size* of the compact elements, as in the proof of Lemma 3.3.

1. Just e. e can be either bottom, number, procedure or *fcontrol*-value. The defining terms for the first three cases are exactly as for Lemma 5.1. If e is an

fcontrol-value, say: $e = \mathsf{inF}(r, k)$. Then, r is a compact value smaller than e. So is the procedure $\mathsf{inL}(\mathsf{inP}(k))$. Using the other cases, let r be definable by R and $\mathsf{inL}(\mathsf{inP}(k))$ by K. Then e itself is defined by:

$$(K\ (\mathit{fcontrol}\ R))$$

2. $(0 \Rightarrow e) \Rightarrow 1$, procedure mapping the thunk form of e to 1, where e is non-bottom, i.e., a number, procedure or *fcontrol*-value.

   For this, we need a predicate in the language that tests if its argument thunk represents an *fcontrol*-application:

   fcontrol-app? $\stackrel{\mathrm{df}}{=}$
   (**lambda** (th)
   (**let** ([x (% (**begin** (th 0) 0) (**lambda** (r k) 0))])
   (**let** ([y (% (**begin** (th 0) 1) (**lambda** (r k) 0))])
   (*equal?* x y))))

   Here, x and y are guaranteed to be numbers, because on normal return, the prompts yield 0 and 1 respectively, and for *fcontrol*-applications, both handlers return 0. Thus x and y are different only for normal return.

   There are now three cases, depending on which subdomain e belongs to:

   (a) (e is a number n.) Define predicates *n?* as defined for Lemma 5.1. Then $(0 \Rightarrow e) \Rightarrow 1)$ is defined by:

   (**lambda** (th)
   (**if** (*fcontrol-app?* th) Ω
   (**let** ([x (th 0)])
   (**if** (**and** (*number?* x) (*n?* x)) 1 Ω))))

(b) (e is a procedure.) Let $E = \{\ldots, a_i \Rightarrow b_i, \ldots\}$. Let Ai define a_i and $Bi?$ define $(0 \Rightarrow b_i) \Rightarrow 1$. Then $(0 \Rightarrow e) \Rightarrow 1$ is defined by:

```
(lambda (th)
  (if (fcontrol-app? th) Ω
     (let ([x (th 0)])
       (if (and (procedure? x)
                ... (Bi? (lambda (d) (x Ai))) ...)
          1 Ω))))
```

(c) (e is an *fcontrol*-value.) Let $e = \mathsf{inF}(r, k)$, where r and $\mathsf{inL}(\mathsf{inP}(k))$ are clearly compact. From the other cases, $(0 \Rightarrow r) \Rightarrow 1$ and $(0 \Rightarrow \mathsf{inL}(\mathsf{inP}(k))) \Rightarrow 1$ are definable, by $R?$ and $K?$ say. Then $(0 \Rightarrow e) \Rightarrow 1$ is defined by:

```
(lambda (th)
  (% (begin (th 0) Ω)
     (lambda (r k)
       (and (R? r) (K? k)))))
```

3. $[(0 \Rightarrow e) \Rightarrow 1] \sqcup [(0 \Rightarrow f) \Rightarrow 0]$, procedure mapping the thunk forms of e, f to $1, 0$ respectively, where e and f are inconsistent. The possible cases are:

   (a) (Both are numbers.) Let e and f be the numbers m and n respectively. Clearly, $m \neq n$: let $m < n$. Then, the defining term is:

$$
\begin{array}{l}
(\textbf{lambda}\ (th) \\
\quad (\textbf{if}\ (\textit{fcontrol-app?}\ \ th)\ \Omega \\
\quad\quad (\textbf{let}\ ([x\ (th\ 0)]) \\
\quad\quad\quad (\textbf{if}\ (\textit{not}\ \ (\textit{number?}\ \ x))\ \Omega \\
\quad\quad\quad\quad (\textbf{if}\ (m?\ \ x)\ 1 \\
\quad\quad\quad\quad\quad (\textbf{if}\ (n?\ \ x)\ 0\ \Omega))))))
\end{array}
$$

(b) (Both are procedures.) Let E And F be the minimal representations of e and f respectively. Since e and f are inconsistent, there must be a $p = p_0 \Rightarrow p_1$ in E and a $q = q_0 \Rightarrow q_1$ in F such that p, q are inconsistent. I.e., p_0, q_0 are consistent, but p_1, q_1 are not.

Let *P0-&-Q0* define $p_0 \sqcup q_0$; *P1-V-Q1?* define $[(0 \Rightarrow p_1) \Rightarrow 1] \sqcup [(0 \Rightarrow q_1) \Rightarrow 0]$; *E?* define $(0 \Rightarrow e) \Rightarrow 1$; and *F?* define $(0 \Rightarrow f) \Rightarrow 1$. Then the following term defines $[(0 \Rightarrow e) \Rightarrow 1] \sqcup [(0 \Rightarrow f) \Rightarrow 0]$:

$$
\begin{array}{l}
(\textbf{lambda}\ (th) \\
\quad (\textbf{if}\ (\textit{fcontrol-app?}\ \ th)\ \Omega \\
\quad\quad (\textbf{let}\ ([x\ (th\ 0)]) \\
\quad\quad\quad (\textbf{if}\ (\textit{not}\ \ (\textit{procedure?}\ \ x))\ \Omega \\
\quad\quad\quad\quad (\textbf{if}\ (\textit{P1-V-Q1?}\ \ (\textbf{lambda}\ (d)\ (x\ \textit{P0-\&-Q0}))) \\
\quad\quad\quad\quad\quad (\textit{E?}\ \ th)\ (\textit{not}\ \ (\textit{F?}\ \ th)))))))
\end{array}
$$

(c) (Both are *fcontrol*-values.) Then $e = \mathsf{inF}(r,k), f = \mathsf{inF}(q,j)$, where r, q, $\mathsf{inL}(\mathsf{inP}(k))$ and $\mathsf{inL}(\mathsf{inP}(j))$ are compact, smaller values. By the other cases, the predicates $(0 \Rightarrow r) \Rightarrow 1$, $(0 \Rightarrow q) \Rightarrow 1$, $(0 \Rightarrow (\mathsf{inL}(\mathsf{inP}(k)))) \Rightarrow 1$, and $(0 \Rightarrow (\mathsf{inL}(\mathsf{inP}(j)))) \Rightarrow 1$ are definable, by *R?*, *Q?*, *K?* and *J?* say. Then $[(0 \Rightarrow e) \Rightarrow 1] \sqcup [(0 \Rightarrow f) \Rightarrow 0]$ is defined by:

$$
\begin{array}{l}
(\textbf{lambda}\ (th) \\
\quad (\%\ (\textbf{begin}\ (th\ 0)\ \Omega) \\
\qquad (\textbf{lambda}\ (r\ k) \\
\qquad\quad (\textbf{por}\ (\textbf{and}\ (R?\ r)\ (K?\ k)) \\
\qquad\qquad\quad (\textbf{and}\ (Q?\ r)\ (J?\ k))))))
\end{array}
$$

where **por** is the form for parallel disjunction:

$$(\textbf{por}\ M\ N) \stackrel{\text{df}}{=} (\textbf{pif}\ M\ 1\ (\textbf{pif}\ N\ 1\ 0))$$

(d) (e and f belong to different subdomains (number, procedure, *fcontrol*-value).) The corresponding defining terms use the predicates *fcontrol-app?* to distinguish *fcontrol*-values from the rest, and then *number?* to distinguish numbers from procedures. After this check, they invoke the terms for either $(0 \Rightarrow e) \Rightarrow 1$ or $(0 \Rightarrow f) \Rightarrow 1$. These, as shown above, are definable, say by $E?$ and $F?$ respectively.

i. (One is an *fcontrol*-value, the other is a number or procedure.) Let e be the *fcontrol*-value. The defining term is:

$$
\begin{array}{l}
(\textbf{lambda}\ (th) \\
\quad (\textbf{if}\ (\textit{fcontrol-app?}\ th) \\
\qquad (E?\ th)\ (\textit{not}\ (F?\ th))))
\end{array}
$$

ii. (One is a number, the other a procedure.) Let e be the number. The defining term is:

$$
\begin{array}{l}
(\textbf{lambda}\ (th) \\
\quad (\textbf{if}\ (\textit{fcontrol-app?}\ th)\ \Omega \\
\qquad (\textbf{let}\ ([x\ (th\ 0)]) \\
\qquad\quad (\textbf{if}\ (\textit{number?}\ x)\ (E?\ th)\ (\textit{not}\ (F?\ th)))))) \blacksquare
\end{array}
$$

Theorem 5.2 The extended direct model $\mathbf{D}_f$ is fully abstract for PS+*fcontrol*+**pif**+%.

Proof The proof follows from Lemma 5.2, using a strategy similar to that of Theorems 3.2 and 5.1. We need to show that if terms M and N have different meanings, then they are observationally distinguishable.

Let $\mathcal{A}'_f[\![M]\!] = m$ and $\mathcal{A}'_f[\![M]\!] = n$ with $m \neq n$. Let us construct a program context $C[\,]$ that distinguishes M and N.

1. The values m and n can be bottom, observable, procedure or *fcontrol*-value. Consider first the case where m and n belong to different categories among these four.

   (a) Distinguishing bottom from observables, procedures and *fcontrol*-values. Let $m = \bot$ and n anything else. Then the empty context distinguishes M and N.

   (b) Distinguishing *fcontrol*-values from observables and procedures. Let m be an *fcontrol*-value and n an observable or procedure. Then the context

   $$(\textit{fcontrol-app?}\ (\textbf{lambda}\ (d)\ [\,]))$$

   distinguishes M and N.

   (c) Distinguishing observables from procedures. Let m be a number and n a procedure. Then the context

   $$(\textit{number?}\ [\,])$$

   distinguishes M and N.

2. Next, we must distinguish between different values in the same category.

(a) Distinguishing between numbers. A context using simple arithmetic operations distinguishes M and N.

(b) Distinguishing between procedures. Since m and n are different procedure values, there must be a precompact procedure element in one—say m—that is not in the other—say n. Let this element be $p = a_0 \Rightarrow \ldots a_k \Rightarrow b$. By Lemma 5.2, the a_i are definable by A_i, and b by B. Then applying the terms M and N respectively to A_0, ..., A_k gives B in the case of M and something else for N, which can then be distinguished using the other cases.

(c) Distinguishing between *fcontrol*-values. Let $m = \mathsf{inF}(r, k)$ and $n = \mathsf{inF}(q, j)$. As above, if m and n are different, there must be a precompact element

   i. in r that is not in q; or

   ii. in q that is not in r; or

   iii. in k that is not in j; or

   iv. in j that is not in k.

   Without loss of generality, let us consider only cases (i) and (iii).

   Let R, Q, K and J define the values r, q, $\mathsf{inL}(\mathsf{inP}(k))$ and $\mathsf{inL}(\mathsf{inP}(j))$ respectively.

   In case (i), let context $C[\ ]$ distinguish R and Q. Then M and N are distinguished by the context:

$$(\%\ [\ ]\ (\textbf{lambda}\ (r\ k)\ C[r]))$$

   Similarly, in case (iii), let context $D[\ ]$ distinguish K and J. Then M and N are distinguished by the context:

$$(\% \; [\,] \; (\textbf{lambda} \; (r \; k) \; D[k]))$$

Thus if $M \not\equiv N$, we can find a program context showing that $M \not\simeq N$. Hence, the model is fully abstract. ∎

Chapter 6

Using control handlers

Control-handling delimiters, or simply *prompts*, in concert with operators for abstracting continuations, have a variety of uses. They provide succinct implementations for a wide range of control paradigms, from simple non-local exits to timer-interruptable computational processes. The following is a collection of some uses of the prompt, for both first- and higher-order abstractions.

In this chapter, we will often use the additional capabilities of real Scheme instead of working within PS. These features include assignment, input/output procedures, and zero- and variable-argument procedures [5].

Sometimes, it is convenient to view the control-delimiting construct as a procedure. This procedure is named *run*, borrowing a term from an operating-system routine that *runs* programs [51]. In a call-by-value language, a *procedural* delimiter like *run*, in contrast to the syntactic form *prompt*, requires its arguments to be values. Thus, *run* takes two argument *values*, the first a *thunk* or procedure of no arguments that encapsulates the prompt's first subexpression, and the second a handler procedure that is the value of the prompt's handler subexpression. A definition for *run* in terms of a primitive form % would be:

$$(\textbf{define}\ run\ (\textbf{lambda}\ (th\ h)\ (\%\ (th)\ h)))$$

On the other hand, given the procedure *run* as the primitive (see Chapter 7), the prompt % can be defined as a syntactic extension:

$$(\%\ e\ h) \equiv (run\ (\textbf{lambda}\ ()\ e)\ h)$$

We shall henceforth use both terms "*run*" and "prompt," preferring "prompt" when emphasizing a program context, and "*run*" when highlighting the use of the delimiter as a *procedure*.

6.1 Simple exits

A prompt with a handler that ignores the continuation—*losing*, rather than *using*, control—simulates the first-order **abort**. The prompt marks the entry/exit point; an *fcontrol*-application within the prompt's first subexpression exits to the entry/exit point with an aborted value. E.g., the following procedure for multiplying the elements of a list exits immediately on a zero element:

```
(define product
  (lambda (s)
    (% (let loop ([s s])
          (if (null? s) 1
              (let ([a (car s)])
                (if (= a 0) (fcontrol 0)
                    (* a (loop (cdr s)))))))
       (lambda (r k) r))))
```

This is a typical example of first-order control manipulation: aborting subcomputations, exiting procedures and loops, and handling basic exceptions. In all these cases, the only control act is a jump to a dynamically enclosing context, perhaps with a throw value—no continuation is required, as the vacuous handler shows. Other higher-order operators like *call/cc* use continuations to mark the entry/exit point, and indeed, need extra code to ensure that (a) such continuations are disabled when they can no longer reflect first-order usage and (b) the garbage collector can reach such disabled continuations [26]. In contrast, the *run/fcontrol* solution mimics first-order aborts precisely.

6.2 Tree-matching

A canonical example of the use of continuations is determine whether two trees have the same *fringe*.[13] The purely functional approach flattens both trees and checks if the results match:

```
(define same-fringe?
  (lambda (tree1 tree2)
    (let loop ([l1 (flatten tree1)] [l2 (flatten tree2)])
      (cond [(and (null? l1) (null? l2)) #t]
            [(or (null? l1) (null? l2)) #f]
            [(eq? (car l1) (car l2)) (loop (cdr l1) (cdr l2))]
            [else #f]))))
```

This procedure traverses the trees once completely to flatten them, and then again till it finds differing leaves. Furthermore, even the best flattening operations require *cons*es equal to the total number of leaves.[14]

[13] In this treatment, ((1 . 2) . 3) and (1 . (2 . 3)) are considered to have the same fringe, as also ((1 2) 3), (1 (2 3)) and ((1 2) (3)). The empty list (), wherever it occurs in the tree, does not contribute any leaves.

[14] For example:

```
(define flatten
  (lambda (s)
    (let loop ((s s) (r '()))
      (cond [(null? s) r]
            [(pair? s) (loop (car s) (loop (cdr s) r))]
            [else (cons s r)]))))
```

The Scheme *call/cc* solution enlists assignment to avoid unnecessary *cons*ing. Each tree is mapped to a *generator*, a procedure with internal state that successively produces the leaves of the tree:

```
(define make-generator
  (lambda (tree)
    (letrec ([caller '*]
             [generate-leaves
               (lambda ()
                 (let loop ([tree tree])
                   (cond [(pair? tree) (loop (car tree)) (loop (cdr tree))]
                         [(null? tree) 'skip]
                         [else (call/cc
                                 (lambda (rest-of-tree)
                                   (set! generate-leaves
                                     (lambda () (rest-of-tree '*)))
                                   (caller tree)))]))
                 (caller '()))])
      (lambda ()
        (call/cc
          (lambda (k)
            (set! caller k) (generate-leaves)))))))
```

The generator returns the empty list (which cannot be a leaf) when all the tree's leaves have been accounted for. A simple loop alternately calls each generator, matches the leaves so obtained, and stops immediately upon finding a mismatch:

```
(define same-fringe?
```

```
(lambda (tree1 tree2)
  (let ([gen1 (make-generator tree1)] [gen2 (make-generator tree2)])
    (let loop ()
      (let ([leaf1 (gen1)] [leaf2 (gen2)])
        (if (eqv? leaf1 leaf2)
            (if (null? leaf1) #t (loop))
            #f))))))
```

The generator procedure uses *call/cc* to keep track of two continuations: (1) the continuation of each call to the generator so the result can be returned to it, and (2) the continuation marking each break in the traversal of the tree, so that the next call to the generator can resume where the previous call left off. The generator's internal variables *caller* and *generate-leaves* record changes in these two continuations. Such assignments can be avoided by having the generators return not just the leaves but also the rest of their computations, but this would require intermediate structures, involving the *cons* overhead we had hoped to avoid.

The crucial continuation is (2), the rest of the computation in the generator. The continuation (1) merely *handles* the interface with the generator. In the *call/cc* solution, each continuation represents a different instance of the *entire* program context. In fact, continuation (1) is used to remember that point in the continuation (2) where control needs to be transferred back to the caller. In the presence of the continuation-delimiting handler, continuation (1) need not be captured at all, and furthermore, continuation (2) need only be the *partial* continuation within the generator. As a side benefit, the entire bookkeeping using assignment can be avoided.

We now present the solution that relies on prompt and *fcontrol* rather than *call/cc* and **set!**. Here too, the program checks the leaves alternately, using generators that successively *throw* leaves:

```
(define make-generator
  (lambda (tree)
    (lambda (any)
      (let loop ([tree tree])
        (cond [(pair? tree) (loop (car tree)) (loop (cdr tree))]
              [(null? tree) '*]
              [else (fcontrol tree)]))
      (fcontrol '()))))
```

A loop uses handlers to *catch* leaves thrown alternately from each fringe, and compares them: a mismatch immediately stops the process:

```
(define same-fringe?
  (lambda (tree1 tree2)
    (let loop ([fringe1 (make-fringe tree1)] [fringe2 (make-fringe tree2)])
      (% (fringe1 '*)
         (lambda (leaf1 rest-of-fringe1)
           (% (fringe2 '*)
              (lambda (leaf2 rest-of-fringe2)
                (cond [(and (null? leaf1) (null? leaf2)) #t]
                      [(or (null? leaf1) (null? leaf2)) #f]
                      [(eqv? leaf1 leaf2) (loop rest-of-fringe1 rest-of-fringe2)]
                      [else #f])))))))))
```

Each time the rest of a fringe is probed, a handler is used to collect a leaf (or the empty list signaling end of fringe) and the remaining fringe computation. If the leaves from the two fringes match, more leaves are ordered. If the leaves are different, the remaining parts of the fringes are ignored, and the predicate returns false.

6.3 All the prefixes of a list

Prompts permit efficient list manipulations that are not possible in a purely functional approach using recursion and *cdr*'ing down the list. For example, the functional program for finding all the non-empty prefixes of a list is:

```
(define all-prefixes
  (lambda (l)
    (if (null? l) '()
      (let ([a (car l)])
        (cons (cons a '())
          (map (lambda (p) (cons a p))
               (all-prefixes (cdr l))))))))
```

An empty list has no prefixes. Given a list of n (> 0) elements, the program recursively finds the prefixes of the tail of the list, adds the head of the list to each of these prefixes to get a prefix of the bigger list, and includes one additional prefix, viz., the single-element prefix containing just the head of the list. For example:

```
(all-prefixes '(1 2 3))
  ⇒ ((1) (1 2) (1 2 3))
```

Let $N(n)$ be the number of *cons*es used by the above procedure for a list of length n. Then, $N(0)$ is 0. For $n > 0$, $N(n)$ is broken up as: (i) $N(n-1)$ for the sublist of the last $(n-1)$ elements, (ii) $(n-1)$ to add the head of the list to each of the prefixes of the sublist, (iii) $(n-1)$ to collect the new prefixes using *map*, (iv) 1 to construct the single-element prefix, and (v) 1 to add the singleton to the rest of the prefixes. Thus, $N(n) = N(n-1) + 2n$, solving which gives $N(n) = n^2 + n$.

On the other hand, the final answer suggests that the only *cons*es required are those needed to construct each of the prefixes (whose length arithmetically increases from 1 to n) and to collect them to a list. This gives $M(n) = n(n+1)/2 + n = (n^2+3n)/2$. The overhead $M(n) - N(n) = n^2 - n$ is due to the wasted *cons*es used to construct the list of prefixes of the argument's tails, only to be destructured when the head is added to each of their elements. In addition to the *cons* overhead, the program spends time destructuring subresults.

The prompt version of the procedure avoids this overhead:[15]

```
(define all-prefixes
  (lambda (l)
    (letrec ([loop (lambda (l)
                     (if (null? l)
                         (fcontrol 'done)
                         (cons (car l) (fcontrol (cdr l)))))])
      (% (loop l)
        (rec h
          (lambda (r k)
            (if (eq? r 'done) '()
              (cons (k '())
                (% (k (loop r)) h)))))))))
```

The *loop* is responsible for determining the actions and argument components required to build the prefix list. On encountering an element, it sends via an *fcontrol*-application the rest of the list and a continuation describing how the prefix up to the element is to be formed. The handler uses the continuation in two ways: (i) to create the prefix up to the element, and (ii) to piggyback this continuation when calling the loop on the rest of the list to get the prefixes beyond the current element. The

[15]The syntax **rec** helps define recursive functions: (**rec** f x) $\equiv$ (**let** ([f '*]) (**set!** f x)).

handler's recursive call to *loop* employs the same handler to collect the result—hence the handler is a recursive procedure. The handler further adds (*cons*es) the current prefix to the longer prefixes that will emerge from action (ii). Action (ii) takes place within a fresh prompt, since the formation of the prefixes should be delimited from the collection of the prefixes into a final list.

This solution does not perform *any* unnecessary *cons*. It also avoids the destructuring of subresults.

6.4 Finding the recombinations of two lists

Another interesting list-manipulating problem is the enumeration of all the *recombinations* of two equal-length lists ("chromosomes"), given the maximum number of *crossovers* allowed. A "recombination" is a list whose ith element ("gene") is the i-th element of either the first or the second of the two argument lists. A "crossover" in a recombination is a position i where the ith and $(i+1)$st elements stem from different arguments.[16] For example:

```
(all-recombinations '(a b c) '(1 2 3) 0)
  ⇒ '((a b c) (1 2 3))
(all-recombinations '(a b c) '(1 2 3) 1)
  ⇒ '((a b c) (a b 3) (a 2 3) (1 2 3) (1 2 c) (1 b c))
```

Armed with prompts, we hereby solve this problem using no extraneous *cons*ing or list-destructuring:

```
(define all-recombinations
  (lambda (l1 l2 crossovers)
    (letrec ([loop
```

[16]The terms come from genetics, where however usually only one crossover takes place.

```
(lambda (l1 l2 n first-preferred?)
  (fcontrol
    (lambda (k)
      (cond
        [(null? l1) (cons (k '()) (fcontrol 'next))]
        [(= n 0)
         (cons (k l1)
           (if first-preferred? (fcontrol 'next)
             (cons (k l2) (fcontrol 'next))))]
        [else
         (let ([a1 (car l1)] [a2 (car l2)]
               [d1 (cdr l1)] [d2 (cdr l2)])
           (% (k (cons a1 (loop d1 d2 n #t)))
              (rec h1
                (lambda (r1 k1)
                  (if (eq? r1 'next)
                      (% (k (cons a2
                                  (loop d2 d1
                                    (if first-preferred?
                                        (- n 1)
                                        n)
                                    #t)))
                         (rec h2
                           (lambda (r2 k2)
                             (if (eq? r2 'next)
                                 (k1 (k2 (fcontrol 'next)))
```

```
                                        (% (r2 k2) h2)))))
                                  (% (r1 k1) h1))))))])))))])
      (% (loop l1 l2 crossovers #f)
        (rec h
          (lambda (r k)
            (if (eq? r 'next) (k '())
              (% (r k) h))))))))
```

The first two arguments of *loop* are the remaining parts of the chromosomes from which the rests of the recombinations are to be built. Initially, the arguments contain the entire input chromosomes. The last two arguments of *loop* keep track of the crossovers: The third argument represents the number of crossovers still permitted, and the fourth is a flag that signals whether the first argument is "preferred," i.e., if successive genes can be extracted from it without signaling a crossover. Initially, neither chromosome is preferred, but once a crossover is made, any further crossovers from *loop*'s first argument should decrement the crossover count.

The procedure *loop* requires two types of control handling as it traverses down its argument chromosomes: (i) to capture the act of building a top fraction of a recombination, and (ii) to capture the act of adding a completely built recombination to the rest of the recombinations that will be built next. For the first, *loop* collects the work of building the top fraction of a recombination into an *fcontrol*-continuation. Since this top fraction forms part of several different answer recombinations, it is placed in two different prompts so that further inner prompts and *fcontrol*-calls include it tacitly. For the second, *loop* uses *fcontrol* to issue a 'next signal: the handler receiving this simply finishes up the current recombination with a suitable bottom fraction and tacks it on to a new computation that will give more recombinations.

The prompts in the body of *loop* launch different subcomputations required for generating the next element in the current recombination, whose prefix is the current *loop*-call's continuation. We can go with either the chromosome favored by the current *loop*-call or cross over to the other chromosome—in either case, recursive calls to *loop* will have larger continuations, each representing a longer top fraction that finally becomes a whole recombination. The first prompt selects a gene from the *loop*'s first argument and recursively calls *loop* to build all recombinations with the current top fraction and that gene. Its handler handles a `'next` request by recursively invoking, in a second prompt, the loop favoring the second chromosome and subtracting one crossover. Thus, between them, the two prompts collect all possible recombinations with the current top fraction. Having done this, the second prompt processes a `'next` request by composing both handlers' continuation arguments before dispatching the `'next` request to *its* enclosing prompt. The composed continuations represents the list of completed recombinations for the *loop*'s arguments.

The first call to *loop* occurs within a prompt. When it finally returns a `'next` request, there are no more recombinations to be built. Thus, the handler dispatches the continuation with an empty list, producing the list of all the recombinations.

6.5 Tagged *run* and *fcontrol*

One inconvenience in using *run* and *fcontrol* as above is that their interrelationship is dynamic rather than lexical. As their use proliferates, we need to cleanly separate pairwise uses of *run*/*fcontrol* so that they do not interfere. Fortunately, this problem is easily solved if we associate *tags* with these control operators, e.g.,

(run^{tagged} `'tag` *thunk handler*) ;or

($\%^{tagged}$ `'tag` *expression handler*)

($fcontrol^{tagged}$ 'tag *throw-value*)

In the following, the names *run*, % and *fcontrol* stand for such tagged operators, with the understanding that a tagged *fcontrol* interacts only with a *run* (or %) with an identical tag. This tagging protocol, among others, is described in further detail in Chapter 7.

The simple abort provided by the handler (**lambda** (r k) r) turns out in the tagged scenario to be identical to the **catch/throw** mechanism of traditional Lisp systems [49], a useful embellishment of the basic procedure/loop exit.

6.5.1 Error handling

A useful special tag (say 'error) may be reserved for user-manipulable error handling. Thus, computations run inside a call to a prompt tagged 'error will throw any errors to the prompt's handler. In other words, the system's error procedure contains an *fcontrol*-application with the tag 'error. The user can employ the 'error prompt to catch errors dynamically at arbitrary points of the program in a manner reminiscent of the pre-Common-Lisp form **errorset** [34].

6.6 Coroutines

A coroutine [27, 32] generalizes a procedure by allowing interruptions and resumptions. Invoking a coroutine is much like invoking a procedure. However, at any point during the execution of the coroutine, a **resume** statement can transfer control to a different coroutine. The suspended coroutine stores its remaining computation as its local control state. On resumption, the coroutine computation proceeds from exactly the point where it was interrupted.

Haynes, Friedman and Wand [27] describe a succinct coroutine implementation that exploits *call/cc*. Each **resume** statement records the current continuation as the coroutine's local control state. However, since this continuation represents the rest of the entire program computation, not just the rest of the coroutine, the *call/cc* solution uses a subtle wrapper around the coroutine call to ensure proper sealing. Handlers, by outrightly delimiting a coroutine's computational context, avoid such scheming:

```
(coroutine x e ...) ≡
  (letrec ([local-control-state (lambda (x) e ...)]
           [resume (lambda (c v)
                     (fcontrol 'coroutine (lambda () (c v))))])
    (lambda (v)
      (% 'coroutine (local-control-state v)
        (lambda (r k)
          (set! local-control-state k)
          (r)))))
```

Here, the coroutine is presented as a unary procedure that calls its *local-control-state* within a prompt. At all resumption points—including the initial one—the variable *local-control-state* holds the remaining coroutine computation: Initially this is the entire coroutine procedure.

A **resume** statement stops the coroutine and exits with a thunk containing the next coroutine call to the coroutine's prompt. The prompt's handler updates *local-control-state* and starts the new coroutine. The latter starts *its* remaining computation, as recorded within *its local-control-state*. All coroutine computations take place within their respective prompts, ensuring that their **resume**s only grab coroutine context.

A more sophisticated coroutine paradigm [7, 27] includes an additional **detach** facility that allows a direct transfer of control back to the point where the coroutine(s) were first called. In a *call/cc* world, **detach** requires remembering the continuation of the first coroutine call across all the encountered coroutines. In our implementation, a **detach** simply aborts a result value to the current coroutine prompt:

... (**let*** ([*local-control-state* ...]
[**resume** ...]
[**detach** (**lambda** (v)
(*fcontrol* 'coroutine (**lambda** () v)))]) ...

6.7 Backtracking through handling

Prolog-style backtracking [6, 50] solves problem or *goal* by trying to solve its *subgoals*. If the goal is a simple or *atomic* goal, it is solved by matching it with statements or *facts* in a database. A goal that is solved is said to succeed.

Given a *query* goal that is a *conjunction* of subgoals, the backtracker checks if each subgoal succeeds. If the query is a *disjunction*, the backtracker checks if at least one of the subgoals succeeds, keeping track of the rest of the subgoals with a *backtrack point*. Should a subgoal fail, the backtracker goes back to the dynamically closest backtrack point to try the next subgoal in *that* disjunction. If all such retries fail, the query as a whole fails.

Implementing backtracking in Scheme provides an apt illustration of continuations. While "purely functional" solutions with goals returning boolean values are possible, such methods require that goals explicitly call success and failure procedures to allow resumption of subgoals at backtrack points. In contrast, Scheme approaches [13, 24] aim for more concise and readable code using *call/cc*-continuations to identify and jump to backtrack points. Control handlers continue this tradition by simply using prompts to mark subgoals.

6.7.1 Unification and logic variables

An *atomic* goal is simply a predicate on terms, where terms are structured objects built from logic variables, numbers, lists and other datatypes. An atomic goal is solved by unifying the term structures composing the goal against facts in the database. (The unification process itself is a predicate: thus, the unification of two terms is an example of an atomic goal.) In this treatment, we will concentrate on the backtracking capabilities provided by control handlers and assume an implementation of both the logic variable datatype and a corresponding unification procedure (refer [13, 24]).

6.7.2 Goals

In this treatment, a goal is a Scheme expression that *throws* (instead of just returning) the boolean *false* if it fails and a true value if it succeeds. In addition, in the latter case, the continuation of the throw represents a backtrack point if the goal is to be retried for an alternate solution. Thus, the "fail" goal is simply:

(*fcontrol* 'goal #f)

The "true" goal is *not*

(*fcontrol* 'goal #t)

but

(**begin** (*fcontrol* 'goal #t) (*fcontrol* 'goal #f))

since it should fail when retried.

A goal is evaluated by running it in a prompt: the handler handles the thrown continuation depending on whether the goal succeeded or failed. The thrown continuation is exactly the rest of the computation of the goal, in other words a representation of the backtrack point in the goal. For example, if == is the unification predicate:

```
(% 'goal (== 1 1) (lambda (r k) r))
```

returns true.

A user query is evaluated like any other goal, viz., inside a prompt: if it succeeds, its logic variables can be examined to see how the query was solved. For example, if the procedures *make-ref* and *deref** make and dereference logic variables respectively, then:

```
(define k0 '*)
(define x (make-ref))
(% 'goal (== 1 x) (lambda (r k) (set! k0 k)))
(deref* x)
```

returns 1, since the logic variable x is bound to 1 by ==. Invoking the backtrack point *k0*:

```
(% 'goal (k0 '*) (lambda (r k) r))
```

returns false, since x has no solutions besides 1.

6.7.3 Disjunction and conjunction of goals

We now define disjunctions (**or!**) and conjunctions (**and!**) as syntactic extensions that take an arbitrary sequence of goals as subexpressions. First, the disjunction:

```
(or! g ...) ≡
  (% 'goal
     (begin
       (% 'goal g
         (rec h
```

```
        (lambda (r k)
          (if r (begin
                  (fcontrol 'goal #t)
                  (% 'goal (k '*) h))))))
      ...
      (fcontrol 'goal #f))
    (rec h
      (lambda (r k)
        (if r (begin
                (fcontrol 'goal #t)
                (% 'goal (k '*) h))
              (fcontrol 'goal #f)))))
```

Each subgoal g is tried successively in a separate prompt. If g fails, its successor is tried, and so on. If, on the other hand, g succeeds, its handler sends a signal of success to the caller of the disjunctive goal. However, g's handler notes that the disjunction should backtrack at g's own backtrack point before trying g's successors. If all the subgoals fail, the disjunction itself fails. This is accomplished by throwing *false* after trying all the goals.

Conjunctions follow a related outline:

```
(and!) ≡ (begin (fcontrol 'goal #t) (fcontrol 'goal #f))
(and! g g2 ...) ≡
  (% 'goal g
    (rec h
      (lambda (r k)
        (if r (% 'goal (and! g2 ...)
                (rec h2
```

```
        (lambda (r2 k2)
          (if r2 (begin
                   (fcontrol 'goal #t)
                   (% 'goal (k2 '*) h2)))
            (% 'goal (k '*) h))))
      (fcontrol 'goal #f)))))
```

The first clause of the definition of **and!** shows that a vacuous conjunction is synonymous with a true goal. If subgoals are present, all of them should succeed for the conjunction to succeed. Each subgoal decides whether the subgoals following it should be tried or not. If a subgoal g succeeds, its *handler* tries the conjunction of the remaining goals, $g2$, etc., but after noting that if these fail, g's own backtrack point should be retried. If g fails, its handler should signal overall failure, without trying g's successors.

6.7.4 The cut

The above implements "pure" Prolog. Often, either for efficiency or a procedural style, we need to prune the backtracking possibilities: Prolog's method is the *cut* ("!"). The *cut* is a goal that succeeds but has the side-effect of committing all the goal choices made from a certain "cut entry" point to the point of the cut. In Prolog, the cut entry is always the immediately enclosing disjunction, but we can relax this restriction here. The syntax **or!!** stands for disjunctions with a cut entry point.

In our implementation, we simply add a handler tagged 'cut at the cut entry point. The expression (**cut**) denotes a cut within this entry point, and is a goal that succeeds at first, but on backtracking, jumps to the cut entry point with a failure signal:

(**or!!** g ...) $\equiv$

```
(let ([cut (lambda ()
               (fcontrol 'goal #t) (fcontrol 'cut '*))])
  (% 'cut (or! g ...)
    (lambda (r k)
      (fcontrol 'goal #f))))
```

6.8 Engines

Implementing the engine abstraction with *run* and *fcontrol* illustrates several facets of the interplay between the two operators. An engine [10, 25] represents computation subject to timed preemption. It forms a tractable building block for realizing a variety of communicating concurrent processes.

An engine's underlying computation is a thunk that can be run as a preemptable process. The engine is applied to three arguments: (1) a number of time units or *ticks*, (2) a *success* procedure, and (3) a *failure* procedure. If the engine computation finishes within the allotted time, the *success* procedure is applied to the result of the computation and the remaining ticks; otherwise, the *failure* procedure is applied to a thunk that represents the rest of the interrupted computation. This thunk, when called, resumes the interrupted engine computation.[17]

Haynes and Friedman [25] distinguish two varieties of engines: *flat* (unnestable) and *nestable*. Flat engines cannot run other engines, but as the authors say, this restriction "considerably simplifies the implementation of engines," where the implementation uses Scheme-style continuations.

The more general nestable engines, or *nesters*, can be called at arbitrary sites, but are more difficult to implement in Scheme. An engine that invokes ("nests") another

[17]Traditionally, the value supplied to the *failure* procedure is a *new engine* representing the remaining computation of the old engine—rather than just its underlying thunk. Our version is no less general, and further allows enhancements that directly access the engine's underlying thunk.

engine is called its *parent*. Nesters require some user-specified notion of *fairness* governing the way time is spent among the nested invocations.[18] For instance, the nestable variety described here lets each engine use ticks only from the amount allotted to its ancestors. Otherwise, an engine could "cheat" by performing its work through its offspring.

The *call/cc* implementation of flat engines involves capture of continuations at both the starting (or resuming) and returning points of an engine. Extending it to allow nestable engines entails more than adding code for tick management, since the continuations to be captured while transferring control across the generations of engines need involved bookkeeping [10].

We show here an implementation of nestable engines using control handlers. There is a clean separation between the segment for transferring control and the segment for managing time units.[19] Indeed, modifying just the time management strategy yields different kinds of fairness, including flat engines.

6.8.1 The clock

The implementation assumes a global clock or interruptable timer that consumes ticks while a program executes. The following describes the type of clock we shall use: it may be defined using either natively provided alarms or through syntactic extensions [10] that simulate tick consumption. The internal state of the clock contains:

1. the number of remaining ticks; and

2. an interrupt handler to be invoked when the clock runs out of ticks.

---

[18]Indeed, the flat engine could be considered a variant of the nester where fairness means the prohibition of children!

[19]Given a module-based Scheme, the code can be written as an engine module that abstracts over a fairness module.

The user can perform the following clock operations:

1. (*clock* 'set-handler $\langle h \rangle$) sets the interrupt handler to $\langle h \rangle$;

2. (*clock* 'set $\langle n \rangle$) sets the ticks for countdown to $\langle n \rangle$; and

3. (*clock* 'stop) stops the clock (without setting off the interrupt handler), returning the remaining ticks.

The number of ticks ranges over the natural numbers and an atom called infinity.[20] A clock with an infinite number of ticks cannot run out of time, i.e., it is quiescent or "already stopped." Stopping an already stopped clock returns infinity. Setting the clock's ticks to infinity stops the clock, i.e., (*clock* 'stop) is shorthand for (*clock* 'set infinity).

The clock's handler is set to throw an interrupt signal, say 'interrupt, to an engine prompt:

```
(clock 'set-handler
  (lambda () (fcontrol 'engine 'interrupt)))
```

6.8.2 The engine core code

The procedure *make-engine* takes a thunk and produces an engine, a procedure of three arguments: *ticks*, *success* and *failure*.

Assume for the moment that the tick management is accomplished by code segments named $\langle$*ticks-prelude*$\rangle$ and $\langle$*ticks-postlude*$\rangle$. The variable *true-ticks*—introduced

[20]Some Scheme dialects provide an atom for an infinitely large number, on which the numerical procedures produce the expected results. In other dialects, any non-numerical atom infinity may be chosen, with the arithmetic procedures *min*, $-$ and $=$ redefined (in the lexical scope of the engine definition) to admit infinity as a possible argument.

in ⟨*ticks-prelude*⟩—shows the actual number of ticks given to the current engine. This may be less than the argument *ticks*, owing to fairness considerations.

When invoked, the engine runs its thunk as an independent piece of computation, in so far as control is concerned. We therefore depict the engine computation as the engine's thunk invoked within a prompt tagged 'engine. The computation uses the flag *engine-succeeded?* to record whether the engine succeeded, and if so, the variable *ticks-left* denotes the ticks to spare. In our first outline, the prompt surrounds code that includes both the initial setting of the clock to the allotted ticks, and the stopping of the clock if the thunk returns successfully. If the engine fails—because of a clock interrupt—the handler returns a thunk representing the rest of the engine. (If the handler was invoked for some reason other than an interrupt, we simply let it pass on the value.)

After the postlude timer code ⟨*ticks-postlude*⟩—which may modify *ticks-left*—either the *success* or *failure* action is taken, depending on the result of running the engine thunk:

```
(define make-engine;; *** first outline ***
  (lambda (thunk)
    (lambda (ticks success failure)
      ⟨ticks-prelude⟩
      (let* ([engine-succeeded? #f]
             [ticks-left 0]
             [result;; ... (I)
               (% 'engine
                  (begin (clock 'set true-ticks)
                    (let ([result (thunk)])
                      (set! ticks-left (clock 'stop))
```

```
            ;; ... (II)
            (set! engine-succeeded? #t)
            result))
          (lambda (r k)
            (if (eq? r 'interrupt)
                (lambda () (k #f)) r)))])
  ⟨ticks-postlude⟩
  ;; ... (III)
  (cond [engine-succeeded? (success result ticks-left)]
        [else (failure result)]))))))
```

When the prompt returns, the variable *result* contains either the rest of the failed engine or a successful result, and the flag *engine-succeeded?* tells which of these is the case. Unfortunately, the code gives incorrect failed engines: the continuation denoting the interrupted engine includes the actions for setting the flag *engine-succeeded?* and stopping the clock. This will yield spurious results when the engine is resumed, whether as a plain thunk or as a fresh engine.

To avoid this, we use *two* prompts. The outer prompt encloses all the computation as before, including the thunk and the clock and flag operations. The new inner prompt surrounds only the setting of the clock and the call to the engine's thunk. The inner handler reacts to interrupts by *throwing* the rest of the engine to the outer prompt, thereby avoiding including the flag and clock operations in the thrown thunk. The outer handler disables interrupts that occur *after* the inner prompt has exited—this is done by resuming the interrupted computation:

```
;;; *** first modification, for (I) above ***
(let* (...
       [result
```

```
(% 'engine
  (let ([result
          (% 'engine
            (begin (clock 'set true-ticks)
              (thunk))
            (lambda (r k)
              (if (eq? r 'interrupt)
                  (fcontrol 'engine
                    (lambda () (k #f)))
                  r)))])
     (set! ticks-left (clock 'stop))
     ;; ... (II)
     (set! engine-succeeded? #t)
     result)
  (lambda (r k)
    (if (eq? r 'interrupt)
        (k #f) r)))])
 ...)
```

A successful engine that finishes with no ticks to spare and suffers an interrupt between the two prompts *could* stop the clock *twice*. To avoid the second stop from setting the number of ticks left to infinity, the latter value must be coerced to zero:

```
;;; *** second modification, for (II) above ***
(set! ticks-left (infinity→0 (clock 'stop))) ...
```

where *infinity→0* is the function (**lambda** (n) (**if** (= n infinity) 0 n)).

The engine currently run may be a child engine, in which case care is needed when invoking the *failure* operations. If the child has no ticks left, the parent may resume with the *failure* action on the rest of the child. If the child does have some ticks left, the child's failure was not because the ticks supplied by the user were insufficient, but because the fairness strategy curtailed its ticks. In the latter case, the parent must resume the child when the parent runs again:

```
;;; *** third modification, for (III) above ***
(cond [engine-succeeded? (success result ticks-left)]
      [(= ticks-left 0) (failure result)]
      [else ((make-engine result)
              ticks-left success failure)]) ...
```

An engine can be forced to stop immediately, either with a success value or as a failure. For a successful exit, it uses *fcontrol* tagged 'engine to transfer control and a success value to the engine prompt:[21]

```
(define engine-return
  (lambda (v) (fcontrol 'engine v)))
```

To block an engine, i.e., compel it to fail, use *fcontrol* to force an interrupt:

```
(define engine-block
  (lambda () (fcontrol 'engine 'interrupt)))
```

[21] For clarity, we have used the symbol 'interrupt for the interrupt signal. In an actual implementation, the signal should be some hidden key to avoid clashing with a potential *engine-return* value that is also 'interrupt.

6.8.3 The code for managing ticks

A flat engine needs very little tick management. The variable *true-ticks*, introduced in ⟨*ticks-prelude*⟩, is set to exactly the *ticks* argument supplied to the engine, since there are no parent engines. Some error-checking to ensure that there is no engine already running may be added:

```
;;; *** ⟨ticks-prelude⟩ for flat engines ***
(if (not (= (clock 'stop) infinity))
    (error 'engine "Trying to nest engines!"))
(let ([true-ticks ticks])
    ...)
```

The ⟨*ticks-postlude*⟩ for flat engines is empty.

For nestable engines, both the prelude and postlude codes are more elaborate. The algorithm first stops the currently active parent engine, if any, before running the new child engine. This yields the ticks left for the parent—infinity if there is no parent engine. For fair nesting, the child cannot be run beyond the parent's remaining ticks, regardless of the ticks allotted to the child in the program. Thus the child should be run for a number of ticks, *true-ticks*, that is the minimum of the parent's remaining ticks and the child's specified ticks. The variable *child-ticks-left* is that part of the child's ticks not accounted for by *true-ticks*, and should be remembered should the child be continued at some later time. Further, the time taken by the child is also counted against the parent—thus, *parent-ticks-left* is the parent's ticks less the child's true ticks:

```
;;; *** ⟨ticks-prelude⟩ for nestable engines ***
(let ([parent-ticks (clock 'stop)]
      [true-ticks (min parent-ticks ticks)]
```

```
    [parent-ticks-left
      (− parent-ticks true-ticks)]
    [child-ticks-left (− ticks true-ticks)])
  ...)
```

In the postlude, both the parent's and the child's remaining ticks are updated to include *ticks-left*, a non-zero number if the child finished successfully before *true-ticks* ran out. The clock is reset to *parent-ticks-left*, thereby restarting the parent engine computation:

```
;;; *** ⟨ticks-postlude⟩ for nestable engines ***
(set! parent-ticks-left (+ parent-ticks-left ticks-left))
(set! ticks-left (+ child-ticks-left ticks-left))
(clock 'set parent-ticks-left)
...
```

The entire *make-engine* code for nestable engines is presented in Figure 6.1.

6.8.4 Engines vs. underlying computation

The above implementation permits a clean separation between an engine's underlying computation and its interruptable version. Thus, we can view the engine as a composite object, consisting of both its thunk and the "engine proper." The procedure *thunk→engine* produces such a composite object from a thunk:

```
(define thunk→engine
  (lambda (th)
    (let ([eng-proper (make-engine th)])
      (lambda (return-either)
        (return-either th eng-proper)))))
```

```
(define make-engine
  (lambda (thunk)
    (lambda (ticks success failure)
      (let* ([parent-ticks (clock 'stop)]
             [true-ticks (min parent-ticks ticks)]
             [parent-ticks-left (- parent-ticks true-ticks)]
             [child-ticks-left (- ticks true-ticks)]
             [engine-succeeded? #f]
             [ticks-left 0]
             [result
              (% 'engine
                 (let ([result
                        (% 'engine
                           (begin (clock 'set true-ticks)
                             (thunk))
                           (lambda (r k)
                             (if (eq? r 'interrupt)
                                 (fcontrol 'engine
                                   (lambda () (k #f)))
                                 r)))])
                   (set! ticks-left (infinity→0 (clock 'stop)))
                   (set! engine-succeeded? #t)
                   result))
                 (lambda (r k)
                   (if (eq? r 'interrupt)
                       (k #f) r))])
        (set! parent-ticks-left (+ parent-ticks-left ticks-left))
        (set! ticks-left (+ child-ticks-left ticks-left))
        (clock 'set parent-ticks-left)
        (cond [engine-succeeded? (success result ticks-left)]
              [(= ticks-left 0) (failure result)]
              [else ((make-engine result)
                     ticks-left success failure)])))))
```

Figure 6.1 The procedure *make-engine* for nestable engines.

The inverse procedure *engine→thunk* extracts the thunk from an engine object:

```
(define engine→thunk
  (lambda (eng-obj)
    (eng-obj (lambda (th eng-proper) th))))
```

Running an engine object involves extracting the engine proper and supplying it ticks, success and failure procedures:

```
(define run-engine
  (lambda (eng-obj ticks succ fail)
    ((eng-obj (lambda (th eng-proper) eng-proper))
     ticks succ fail)))
```

6.9 Summing up

We have shown a wide array of applications for control handlers, ranging from simple exits through control-based (rather than accumulator- or assignment-based) approaches to data manipulation, error handling, coroutines to arbitrarily nestable preemptable tasks. The following chapter will discuss the implementation of these control primitives, together with details on *tagging* them.

Chapter 7

Implementing control operators and delimiters

Implementing any control operator requires some representation of the control context that can be manipulated. At any point during the evaluation of the program, the control context consists of a sequence of frames or activation records of procedure calls that constitute the rest of the computation. Languages with control operators allow the user to capture and use a packaged form of this context information, and the package is called the "continuation."

In this chapter, we will use Scheme's operator *call/cc* as a basis, and discuss how existing machinery for it may be converted to yield *run* and *fcontrol*. The conversion may either take the form of modifying the language's code generator, or use the primitive *call/cc* as a building block to define or *embed* the new operators in the language. The first method alters the language but allows *run* and *fcontrol* as efficient primitives—we call this a "native" implementation.[22] The embedding approach, although similar in spirit, is necessarily circuitous—and therefore inefficient—since its only window to the underlying context information is *call/cc*. It is nevertheless the easier way to add *run* and *fcontrol* to a given Scheme implementation.

7.1 A strategy for native *run* and *fcontrol*

Typically, the sequence of frames comprising the context information of an evaluating program is represented by a *control stack*, which is empty when the program commences, and grows and shrinks as each subcomputation in the program starts and

[22]A "native implementation" of a facility incorporates the facility into the code generator of the system: it allows optimizations that would not be possible for facilities built on top of the system.

completes respectively (Figure 7.1(a)). In a language where control manipulations, if any, are only first-order, no stack increment need survive the end of its corresponding subcomputation, which is why the context information is allocated as a *stack*. Stacks may still be used in a higher-order control scenario like Scheme, if we view higher-order control objects as copies or "snapshots" of the stack (Figure 7.1(b)). Alternatively, recognizing that in such a language these control objects—like the other objects in the language—must have unlimited extent, we can simply store control information as a tree on the heap. This obviates the need for copying whenever a control object is built, since heap objects, unlike stack objects, exist forever.[23]

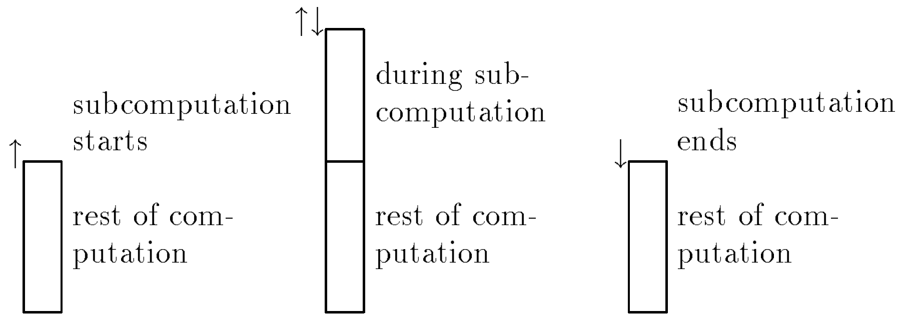
In any case, both stack- and heap-based implementations of *call/cc* package the *entire* control stack into a continuation object for the user. Refining this approach, as in Figure 7.2, to handle any *contiguous portion* of the stack leads to a native implementation of *run* and *fcontrol*.

In such an implementation, a *run*-invocation marks the top of the current control stack with its handler. An *fcontrol*-application captures the portion of the control stack from the top down to its closest *run*-mark—this is the functional continuation. It then erases this portion off the stack, and sends both continuation and the *fcontrol* argument to the handler associated with the *run*-mark. The handler procedure may eventually apply the functional continuation, at which point the abstracted portion of the stack is installed on top of the current control stack.

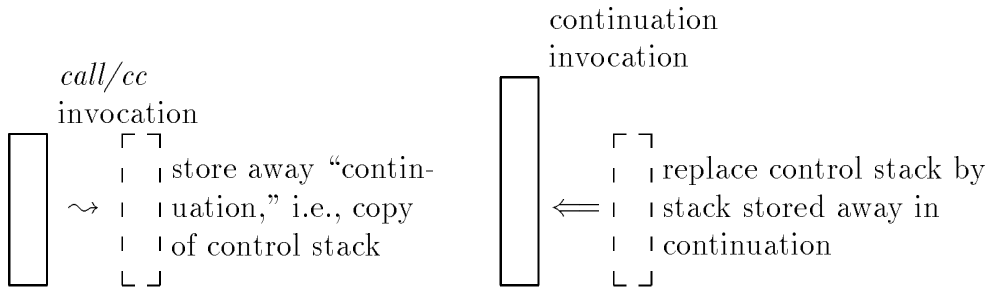
7.2 Embedding *run* and *control* in Scheme

The native implementation demonstrates the difference in the way *run* and *fcontrol* on the one hand and *call/cc* on the other manipulate the control stack. Since the

[23]Of course, not all objects and certainly not all contexts warrant this immortality. To prevent them from taking up space, Scheme implementations invariably include a prudent garbage collector that hunts down objects that are guaranteed not to be used and reclaims their storage area [1, 2, 5].



(a) The control stack at the start of, during, and at the end of a subcomputation.



(b) *call/cc* and the control stack.

Figure 7.1 The control stack.

locus of the difference is the control stack, and since *call/cc* is a *higher*-order control manipulator, we can *simulate run* and *fcontrol* using *call/cc*. The simulation serves as plug-in Scheme code that adds *run* and *fcontrol* to any Scheme implementation. Furthermore, its intricacy is a metric of the difference between the two styles of control manipulation.

First, the underlying control stack or at least a sufficient skeleton thereof has to be made explicit: this is done by building a *run-stack* consisting of *call/cc*-continuations at appropriate *run*-points. Second, this explicit stack should allow the operations required of *fcontrol*: for this, we will associate additional *call/cc*-continuations *with*

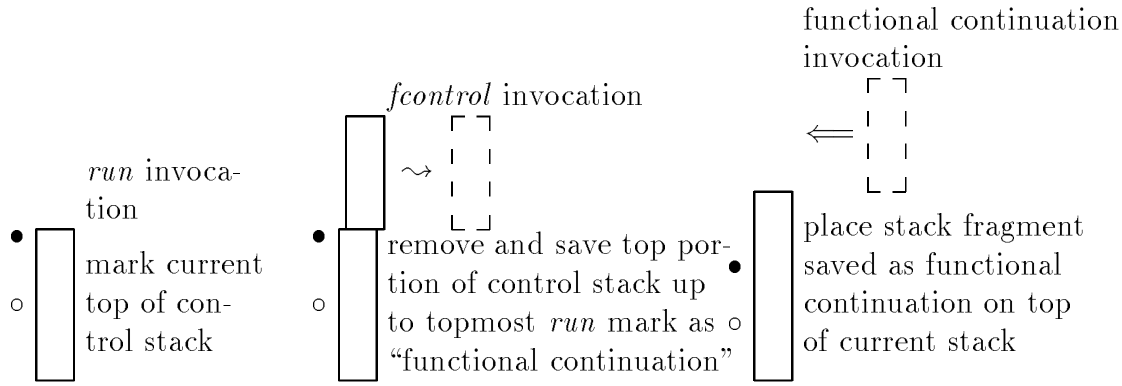


Figure 7.2 *Run*, *fcontrol* and the control stack.

each run-point in the *run-stack*. These extra continuations, as we shall see later, correspond to invocations of functional continuations. They record the calling context of the functional continuation so that control can return there after the partial context to the corresponding *run* has been traversed. In other words, in a system where continuations are abortive, storing the continuation of a *call* to a continuation simulates "functional" continuations, and the explicit stack stores these extra continuations so that the procedure for *run* can ensure that these functional returns are always carried out.

The representation of the underlying control stack is only as grainy as is needed for the particular program. Each *frame* in the *run-stack* corresponds to a *run* (and not an activation record), and contains the *call/cc*-continuation at the *run* as well as a substack of *call/cc*-continuations corresponding to functional-continuation invocations within this *run*. Initially, *run-stack* is empty:

(**define** *run-stack* '())

Each *run*-application captures its *call/cc*-continuation and pushes a frame containing this continuation and an empty substack atop the *run-stack*. If the *run*-argument returns normally with a value, the *run-stack* is popped and the value returned.[24] Thus, the outline of the procedure *run* is:

```
(define run
  (lambda (th hdlr)
    (let* ([run-frame 'any]
           [v (call/cc
                (lambda (k)
                  (set! run-frame (cons k '()))
                  (set! run-stack (cons run-frame run-stack))
                  (th)))])
      (cond
        ... ;did not return normally
        [else ;did return normally
          (set! run-stack (cdr run-stack))
          v]))))
```

The frame *run-frame* pushed by a *run*-application is a list with the *run*'s continuation as its first element. The rest of the components in a frame forms the *run*'s substack. Initially the substack is empty, but this will change if the *run*'s argument thunk calls *fcontrol*.

At any point within a *run*-computation, the current *call/cc*-continuation *and* the topmost substack on the *run-stack* completely determine the remaining computational context that must be traversed before the *run*-computation runs its course. An *fcontrol*-application does two things:

[24] The necessary updating of the *run-stack* unfortunately implies that this embedding is not tail-recursive. Thus, whereas (**let** *loop* () (*run loop*)) is a tight loop in a system with native *run* and *fcontrol*, it will eventually exhaust control stack space in our embedding.

1. It *aborts* to its nearest *run*. For this, it should not only invoke this *run*'s *call/cc*-continuation, but also erase all the auxiliary continuations in the topmost *run-frame*. This is accomplished by removing all the continuations from the *run-frame*'s substack.

2. It sends to the *run*'s handler its throw-value and a packaged version of its context—i.e., the *call/cc*-continuation of the *fcontrol*-application *and* the continuations removed from the topmost substack.

This package of *call/cc*-continuations is called the "functional continuation," and is a procedure that installs and invokes the packaged continuations in the right order. The following outlines the code for *fcontrol*:

```
(define fcontrol
  (lambda (val)
    (call/cc
      (lambda (fctl-cont)
        (let ([fctl-frame (car run-stack)])
          (let ([fctl-run-cont (car fctl-frame)]
                  [fctl-substack (cdr fctl-frame)])
            (set-cdr! fctl-frame '())
            (fctl-run-cont
              (list 'fcontrol-message val
                (lambda (v)
                  ... ;functional continuation formed out of
                  fctl-substack ... ;and
                  fctl-cont ...)))))))))
```

When an *fcontrol*-message arrives at the *run*, the handler is applied to the message's contents:[25]

```
... ; body of run
(let ([v ...])
  (record-evcase v
    ['fcontrol-message (fctl-val fun-cont)
      (set! run-stack (cdr run-stack))
      (hdlr fctl-val fun-cont)]
    [else ...]))
```

The functional continuation created by *fcontrol* remembers both the continuation of the *fcontrol*-application and the substack of invocation points that was on the top of the *run-stack* during the call to *fcontrol*. Upon invocation, this functional continuation must push its own invocation point on to the current topmost substack, so that control can return to its context. The functional continuation then reinstalls its remembered substack on top of the substack, and jumps to the *call/cc*-continuation associated with the *fcontrol*-call:

```
... ;body of fcontrol—the making of the functional continuation
(lambda (v)
  (call/cc
    (lambda (invoke-cont)
```

[25]The syntax **record-evcase** is a simple **case**-like conditional that uses the *first* element of a value (record or list) to decide among a group of clauses. The form successively compares each clause's tag against the value's first member. The first clause that succeeds evaluates its body after binding the rest of the destructured value to lexical variables. If all clauses fail or if the value is not a list, a final **else**-clause, if present, is picked. Unlike **case**, **record-evcase** does not implicitly quote its clauses' tags. This allows lexically hidden tags: Indeed, all the message tags used in this implementation can and should be replaced by such hidden tags to prevent interference with the symbols in a user program.

```
(let ([invoke-frame (car run-stack)])
  (let ([invoke-substack (cdr invoke-frame)])
    (set-cdr! invoke-frame
      (append fctl-substack
        (cons invoke-cont invoke-substack)))
    (fctl-cont v))))))
```

On applying a functional continuation, a *call/cc*-continuation associated with the corresponding *fcontrol*-call is invoked. Thus, control will eventually reach the *run*-context surrounding the *fcontrol*-call, viz., the context enclosing the *capture* of the functional continuation. Since this may in general differ from the topmost *run*-context, viz., the one enclosing the *invocation* of the functional continuation, we add code to the body of the procedure *run* that checks if the *run*-argument has returned in the proper *run*-context, and if not, jumps to the topmost *run*:

```
... ;body of run
(record-evcase v
  ['fcontrol-message ...]
  [else (let* ([top-frame (car run-stack)]
               ...)
          (cond [(not (eq? run-frame top-frame))
                 ((car top-frame) v)]
                ...
                [else (set! run-stack (cdr run-stack))
                  v]))])
```

The *run* also needs to dispatch control back to each of the invocation points in its frame so that *fcontrol*'s continuations are "functional":

```
... ;body of run
(let* ([top-frame (car run-stack)]
       [top-substack (cdr top-frame)])
  (cond ...
    [(not (null? top-substack))
     (set-cdr! top-frame (cdr top-substack))
     ((car top-substack) v)]
    ...))
```

Figures 7.3 and 7.4 collects the above code fragments into the final definitions for *run* and *fcontrol*. The code for *fcontrol* includes some error handling for the cases when an *fcontrol*-application or a continuation invocation is attempted when there is no surrounding *run* to delimit them.

For convenience, the implementation may also provide a thunk *reset-scheme-with-run-and-control* that flushes *run-stack* and spawns a new *read-eval-print* loop. This interactive loop iteratively reads an input expression, surrounds it with a catch-all outermost *run* (called *base-run*), and evaluates it:

```
(define reset-scheme-with-run-and-control
  (lambda ()
    (set! run-stack '())
    (let read-eval-print-loop ()
      (write "% ")
      (write (eval `(base-run (lambda () ,(read)))))
      (newline)
      (read-eval-print-loop))))
```

```
(define run
  (lambda (th hdlr)
    (let* ([run-frame 'any]
           [v (call/cc
                (lambda (k)
                  (set! run-frame (cons k '()))
                  (set! run-stack
                    (cons run-frame run-stack))
                  (th)))])
      (record-evcase v
        ['fcontrol-message (fctl-val fun-cont)
         (set! run-stack (cdr run-stack))
         (hdlr fctl-val fun-cont)]
        [else (let* ([top-frame (car run-stack)]
                     [top-substack (cdr top-frame)])
                (cond [(not (eq? run-frame top-frame))
                       ((car top-frame) v)]
                      [(not (null? top-substack))
                       (set-cdr! top-frame (cdr top-substack))
                       ((car top-substack) v)]
                      [else (set! run-stack (cdr run-stack))
                        v]))]))))
```

Figure 7.3 Embedding *run* in Scheme.

The identifier *base-run* is bound to a procedure that runs its only argument within some suitable handler. It may be modified to reflect small changes in the control strategy, e.g., including checks for special control messages.

7.3 Tagged *run* and *fcontrol*

Multiple uses of *fcontrol* and *run* can interfere with each other. For example, higher-level abstractions such as *coroutines* and *engines defined* using these control operators (Chapter 6) may not be used together, as the particular pairing of instances of *fcontrol*

```
(define fcontrol
  (lambda (val)
    (call/cc
      (lambda (fctl-cont)
        (if (null? run-stack) (error 'no-surrounding-run))
        (let ([fctl-frame (car run-stack)])
          (let ([fctl-run-cont (car fctl-frame)]
                [fctl-substack (cdr fctl-frame)])
            (set-cdr! fctl-frame '())
            (fctl-run-cont
              (list 'fcontrol-message val
                (lambda (v)
                  (call/cc
                    (lambda (invoke-cont)
                      (if (null? run-stack) (error 'no-surrounding-run))
                      (let ([invoke-frame (car run-stack)])
                        (let ([invoke-substack (cdr invoke-frame)])
                          (set-cdr! invoke-frame
                            (append fctl-substack
                              (cons invoke-cont invoke-substack)))
                          (fctl-cont v))))))))))))))
```

Figure 7.4 Embedding *fcontrol* in Scheme.

and *run* is spoiled. This is because *run* and *fcontrol* interact dynamically rather than lexically, and there is no straightforward way to ensure that pairwise uses of *run*/*fcontrol* are cleanly separated.

The natural solution calls for matching pairs of *fcontrol* and *run*. As in the first-order **catch** and **throw**, we can have versions of *run* and *fcontrol* take *tags* as arguments, where a tag is an arbitrary Scheme object. This completes our view (Chapter 2) that *run* and *fcontrol* are straightforward generalizations of their first-order counterparts.

One tagging protocol—others are possible—is to have each matching pair of *fcontrol* and *run* interact within itself and ignore other pairs. We require that a *fcontrol* with a certain tag X jump to its dynamically closest enclosing *run* tagged X. Not only should intervening *run*s with other tags ignored, but the continuation thrown to the X-*run* should be the complete continuation extending from the X-*run* to the X-*fcontrol*-application. Different tags govern different logical uses of *run*/*fcontrol* without fear of interference. Furthermore, since a tag is any object, we can choose unforgeable tag values and hide their use within a textual region using lexical binding. The following defines the tagged operators in terms of the primitive *run* and *fcontrol*.

The procedure $\mathit{fcontrol}^{tagged}$ uses *fcontrol* to send a record containing its tag, its throw value and an identity continuation[26] to its enclosing *run*. The continuation component of this record serves as a base continuation that is composed with *fcontrol*'s functional continuation to get the true continuation:

(**define** $\mathit{fcontrol}^{tagged}$
  (**lambda** (*val tag*)
    (*fcontrol* (*list tag val identity*))))

The procedure run^{tagged}, on receiving the *fcontrol*-message, extracts from it the tag, throw-value and base continuation. It composes the functional and the base continuations to get the true continuation. If the tags of the message and run^{tagged} match, the handler is invoked on the value and the true continuation. Otherwise, it passes along a changed form of the *fcontrol*-message to the next enclosing *run*, with the current true continuation as the new base continuation:

(**define** run^{tagged}
  (**lambda** (*run-tag th hdlr*)

[26] Since continuations are functional, this is just the identity procedure, (**lambda** (x) x).

```
(run th
  (lambda (fctl-msg fun-cont)
    (let ([fctl-tag (car fctl-msg)]
          [fctl-val (cadr fctl-msg)]
          [fctl-cont (compose fun-cont (caddr fctl-msg))])
      (if (eq? fctl-tag run-tag)
          (hdlr fctl-val fctl-cont)
          (fcontrol (list fctl-tag fctl-val fctl-cont)))))))
```

Throws to non-existent tags will eventually arrive at the outermost *run* of the program, e.g., the *read-eval-print* loop. Error-handling of choice may be added here.

With this as well as the other tagging protocols described below, it is preferable to provide the chosen tagged operators as the primitives, by incorporating the tagging protocol directly into the code generator or the *call/cc*-based implementation. In such enhanced systems, the names *run* and *fcontrol* can be usurped for the tagged operators—the system's procedure-definition and syntactic-extension facilities can resolve untagged occurrences of either operator (as in Chapter 6) to some default tag, e.g., *false*.

7.4 Multiply tagged *run* and *fcontrol*

Total independence between the *run*/*fcontrol* pairs may not always be desirable. In such cases, some form of hierarchical interrelationship may be required on several levels of control pairs, e.g., a *run* at a certain level serves as control delimiter not only to the *fcontrol*s of its own level but of all levels below it. Plain *run* and *fcontrol* can build such control hierarchies. Several variations on this theme are possible [46].

As an example, we show here one approach to the problem of selectively interacting control operators: *multiple* tags. From several possible modes of interplay, we choose the following scenario: Both *run* and *fcontrol* can take multiple tags. A *run* delimits an *fcontrol*-application within it if they have a common tag. Additionally, zero tags are considered special for convenience: A *run* with no tags delimits *all fcontrol*-applications within it, regardless of their tags. An *fcontrol*-application with no tags is delimited by any surrounding *run*, regardless of the latter's tag. The definitions in terms of raw *run* and *fcontrol* closely parallel the singly tagged versions. The procedure $fcontrol^{multiply\text{-}tagged}$ now sends all of its tags (if any) to the surrounding *run*:

```
(define fcontrol^multiply-tagged
  (lambda (tags val)
    (fcontrol (list tags val identity))))
```

The procedure $run^{multiply\text{-}tagged}$ includes tests for the three cases:

1. either the *fcontrol*-message or the *run* itself have no tags;
2. the *fcontrol* and *run* have a common tag;[27]
3. neither of the above, in which case the *fcontrol*-message is forwarded—after updating the continuation—to the next surrounding *run*.

```
(define run^multiply-tagged
  (lambda (run-tags th hdlr)
    (run th
      (lambda (fctl-msg fun-cont)
        (let ([fctl-tags (car fctl-msg)]
              [fctl-val (cadr fctl-msg)]
```

[27](*there-exists?* ⟨*list*⟩ ⟨*predicate*⟩) returns *true* iff ⟨*predicate*⟩ is true of at least one element in ⟨*list*⟩.

```
    [fctl-cont (compose fun-cont (caddr fctl-msg))])
  (if (or (null? run-tags) (null? fctl-tags)
          (there-exists? run-tags
            (lambda (run-tag)
              (there-exists? fctl-tags
                (lambda (fctl-tag)
                  (eqv? fctl-tag run-tag))))))
    (hdlr fctl-val fctl-cont)
    (fcontrol
      (list fctl-tags fctl-val fctl-cont))))))))
```

7.5 Postludes

A further enhancement for getting judicious interference between different tagged uses of *run* and *fcontrol* is to associate *postludes* with each delimiter. In programming styles making heavy use of control manipulation, postludes maintain the integrity of local state as control jumps across far-off points. Postludes are used in both first-order and higher-order control arenas. Some well-known forms for invoking postludes are Common Lisp's **unwind-protect** [49] and Scheme's **dynamic-wind** [26]. The latter form also allows *preludes*, since local state needs to be safeguarded on both *incoming* as well as *outgoing* *call/cc*-jumps.

In our case, an illustrative example of the need for postludes is the **engine** code in Chapter 6. An engine's *run* should intercept *all* control operations—not just the ones tagged 'engine. Otherwise, the engine code is not *fault-tolerant*, for the timer would be left running even though the engine has technically exited. In this particular case, the problem is solved by associating a postlude with the engine's *run* so that *every* non-engine exit takes care to stop the clock.

Modifying our previous code for run^{tagged} to accommodate postludes is easy:

```
(define run^{tagged-with-postlude}
  (lambda (run-tag th hdlr postlude) ;⇐
    (run th
      (lambda (fctl-msg fun-cont)
        (let ([fctl-tag (car fctl-msg)]
              [fctl-val (cadr fctl-msg)]
              [fctl-cont (compose fun-cont (caddr fctl-msg))])
          (if (eq? fctl-tag run-tag)
              (hdlr fctl-val fctl-tag)
              (begin (postlude) ;⇐
                (fcontrol (list fctl-tag fctl-val fctl-cont)))))))))
```

The code for $fcontrol^{tagged}$ needs no change.[28]

Armed with the new tagged *run*, we embellish our engine's *run* by adding a postlude that stops the clock. However, this leaves open the question of whether this kind of exit should signify success or failure. Going back to the rationale behind engines, we see that an engine computation is *an independent task as far as control is concerned*. Thus it *seems* spurious to allow any exit from the engine that is not specified by the engine code. Hence, we could have the postlude also signal an error, after winding down the clock. So, the postlude of the engine's inner *run* reads:

```
(lambda ()
  (set! ticks-left (clock 'stop))
  (error "illegal exit attempted from running engine"))
```

[28]Note that we specify that the postlude is executed *only* on a non-matching tag. The Lisp tradition is to execute the postlude uniformly on all exits, including "normal" return. A simple procedure definition can retrieve the traditional variant.

On the other hand, *some* uses of the engine might entail legitimate exits outside the engine computation. For example, consider the following implementation of McCarthy's **amb** (ambiguous or nondeterministic parallel **or**) operator [33] using engines:

```
(amb e ...) ≡
  (let ([q (make-queue)])
    (enqueue! q (make-engine (lambda () e))) ...
    (let loop ()
      (if (queue-empty? q) #f
        ((dequeue! q) *slice-of-time*
         (lambda (v n)
           (or v (loop)))
         (lambda (th)
           (enqueue! q (make-engine th))
           (loop)))))))
```

Thus, an **amb**-expression constructs engines out of its subexpressions and cyclically[29] executes each of them for a slice of time until either one of them returns *true* or all of them turn out *false*.

Now, consider an expression such as:

```
(% 'abort (... (amb ... (... (fcontrol 'abort v) ...) (% ...) ...))
    (lambda (r k) r))
```

[29]The code assumes the availability of a rudimentary *queue* datatype with the usual operations; **slice-of-time** is the number of ticks granted to each member of the queue.

Here, one of the **amb**-subexpressions aborts to a point outside the **amb**. In such a case, the engine's success or failure becomes irrelevant. What is required is an abandoning of the **amb** computation, not an error signal.

To allow for such special treatments of tags, we have the postlude take a tag argument so that it can decide what approach to take. This requires a further modification to run^{tagged}:

```
(define run^{tagged-with-postlude}
  ...
  (if (eq? fctl-tag run-tag)
      (hdlr fctl-val fctl-tag)
      (begin (postlude fctl-tag) ;⇐ instead of just (postlude)
        (fcontrol (list fctl-tag fctl-val fctl-cont))))
```

The engine postlude now becomes:

```
(lambda (tag)
  (set! ticks-left (clock 'stop))
  (if (not (eq? tag 'abort))
      (error "illegal exit attempted from running engine")))
```

Chapter 8

Conclusion

Semantics of various kinds are devised to better capture the behavior of a language. However, they can also be used to pave the way for the introduction of newer, more versatile and more expressive language constructs. This work illustrates a complete instance of such feedback, using a variety of denotational models for languages that allow control manipulation. As we have seen, the theoretical results mesh well with practical needs in producing a workable design principle.

8.1 Model as advisor

Ordinarily, denotational models are reasoning tools for an already designed language; this work shows that they also provide important information on how to design—i.e., improve or enhance—the language.

To get such information, we had to analyze the relationship between the equational theory of the denotational model and the observational theory based on the behavior of whole programs. The model is fully abstract for the language if the two theories match. If not, studying the cause of mismatch with a view to salvaging full abstraction can lead to the discovery of useful language extensions. In essence, we use the full abstraction criterion as a feedback loop—changes are made to the language until the model becomes fully abstract.

In our case, we studied two broad classes of models for languages that allow manipulation of the control context. We confirmed that (1) both first- and higher-order control manipulation require a control delimiting mechanism, and (2) higher-order control requires a handling mechanism that operates on both aborted objects as well as reified context information. The results are useful and pragmatically feasible, and solve many current problems of control programming.

8.2 Contemporary work

There has been a significant amount of recent related work in the area we have just explored. Sieber [44], who identified and corrected an error in my earlier investigation in the full abstraction of domains for $\mathrm{PCF_V}$ (the typed λ-calculus with arithmetic), has presented a general method for relating full abstraction results across various languages. In particular, he has presented a full abstraction result for the typed cps model and $\mathrm{PCF_V}$+*call/cc*+% as a corollary of the theorem for plain $\mathrm{PCF_V}$. The method also promises to serve well for newer language extensions.

Riecke's recent work [42] shows that the delimiter approach may also be useful in languages with imperative features *other than control*. His *effects delimiters* allow separation of the purely functional parts of a program from those using state, control or exceptions. These delimiters can declare and enforce all access to a code fragment to be purely functional—i.e., the delimited fragments, which are observationally equivalent as functional language terms, continue to be so in the extended language. Furthermore, these declarations are valuable in enabling compiler optimizations and proving correctness.

Several control alternatives that go beyond Scheme's *call/cc* have been recently proposed. All of them involve control delimiting and some form of "composable" or "partial" continuations. Danvy and Filinski's **shift** and **reset** [8] and Queinnec and Serpette's *splitter* [40] are two such control proposals for a sequential language: The operator **reset** delimits the context reified by **shift**; the *splitter* operator delimits a computation, and provides it a fresh *call/cc*-like operator for internal use. Both approaches underscore the elegance and pragmatic utility of working with delimited contexts. Hieb, Dybvig and Anderson's *spawn* [28, 29] presents control manipulation in the presence of tree-structured concurrency. The *spawn* operator delimits a *subcomputation* and provides it a controller, a *call/cc-* or *fcontrol*-like operator that reifies "subcontinuations" and permits nonlocal exits to arbitrary points in a tree of processes.

An interesting question is whether we can get analogous results for other language extensions: notably assignment, either by itself or in concert with control extensions. Filinski's [21] simulation of composable continuations with non-composable ones using a single storage cell, as also our Scheme embedding of handlers and *fcontrol* (Chapter 7) suggest an interesting interference between the extensions for advanced control and assignment. It would also be useful to study if handlers, like Riecke's effects delimiters, generalize to non-control extensions.

8.3 Summary

We have use denotational models to understand and design an important if difficult aspect of modern higher-order languages—viz., the apparatus for manipulating nonlocal control. Using information from the failure of full abstraction to strengthen the language provides valuable insight in charting the control map of the language. In particular, the control-delimiting and control-handling mechanisms suggested naturally by the various denotational models provide a powerful and flexible approach to using control well.

Appendix A

Just enough domain theory

Given that a domain is a collection of all the objects of discourse, it still remains to describe these objects. Typically, these objects are interrelated by a potentially complex approximation relation. Some of them are beyond finite description. Hence it would be helpful to have more primitive means of characterizing a domain than just enumerating its contents.

A.1 Information systems

To this end, we follow the information systems approach propounded by Scott [43]. Each object in a domain is specified by the set of propositions that are true of it: we thus move from discussing objects to discussing propositions about objects. An *information system* (I.S.) $\mathcal{D}$ is a collection of *atomic*[30] propositions p, with a relation *entails* ($\vdash$) over them. A trivial proposition, present in every information system, is $\Delta \equiv$ "it is true"—the *least informative* proposition that is true of any object. Other example propositions are "it is round and red", "it is round" and "it is red", which are only true of *some* objects. The first one entails the second and third, and is, in fact, a *conjunction* of the two (and therefore, nonatomic). These last two, however, are not related by the "entails" relation. Further, unless we intend to define roundness and redness by more finely grained properties, these propositions are atomic. (Every proposition entails Δ.)

[30]Contrary to Scott [43], conjunctions of atomic propositions are *not* treated as elements of $\mathcal{D}$.

The three propositions above are *consistent*, i.e., they *could* all hold simultaneously for an object, e.g., an apple. However, the propositions "it is red" and "it is green" are *inconsistent.* Given a consistent set of propositions X, we define its *deductive closure* X° as the set of all the propositions entailed by X—i.e., $\mathsf{X}^\circ = \{\mathsf{p} \mid \mathsf{X} \vdash \mathsf{p}\}$.

An object of the *domain* $\mathbf{D}$ corresponding to an I.S. $\mathcal{D}$ is *defined* as precisely the set of all propositions of $\mathcal{D}$ that are true of it. Clearly, an object is a deductively closed set. And since we are *defining* objects (from whole cloth, as it were), this means that every deductively closed set is an object. There is a least object or *bottom* ($\perp$), defined as the deductive closure of $\{\Delta\}$ (or $\emptyset$).

Since we can invoke deductive closure to fill in the missing propositions, an object may be fully described by just a few of its constituent propositions. Eliminating every proposition that could be deduced from the rest, we arrive at a minimal set of propositions—which we call a *minimal representation*—describing the object. Thus, if M is a minimal representation of object d, then $d = \mathsf{M}^\circ$. (Note that minimal representations need not be unique, since the order of elimination can yield different final results.) We call objects with finite minimal representations *compact.*[31]

A natural approximation ("below"ness) order $\sqsubseteq$ on the elements of the domain is simply the subset ($\subseteq$) relation: thus we can use $\sqsubseteq$ and $\subseteq$ interchangeably. An *upper bound* (ub) of a set of domain objects is a domain object that contains them all. The *least* upper bound (lub: $\sqcup$) is the smallest such bound. Clearly, the lub is constructed by taking the deductive closure of the union ($\bigsqcup S = (\bigcup S)^\circ$).

Theorem A.1 A domain element is the *union* of all the compact elements below it.

[31]Compact elements are also called "finite" objects, but this could be confusing, since the deductive closure of a minimal representation may have an infinite number of propositions.

Proof Let $D = \{\ldots, d_i, \ldots\}$ be the compact elements below d, i.e., $d_i \sqsubseteq d$ for all d_i.

To show that $\bigcup D$ and d are subsets of each other.

First, to show that $\bigcup D \subseteq d$.

> Let $\mathsf{p} \in \bigcup D$.
>
> Thus, $\mathsf{p} \in d_i$ for some i.
>
> Since $d_i \sqsubseteq d$, $\mathsf{p} \in d$.

Next, to show that $d \subseteq \bigcup D$.

> Let $\mathsf{p} \in d$.
>
> Let M be a minimal representation of d, i.e., $d = \mathsf{M}^\circ$.
>
> Thus, $\mathsf{M} \vdash \mathsf{p}$.
>
> I.e., *finite* $\mathsf{M}' \vdash \mathsf{p}$, for some $\mathsf{M}' = \{\ldots, \mathsf{m}_j, \ldots\} \subseteq \mathsf{M}$.
>
> I.e., $\mathsf{p} \in \mathsf{M}'^\circ$.
>
> But, M'° is a compact element below d.
>
> Therefore, it is a d_i for some i.
>
> Therefore $\mathsf{p} \in \bigcup D$. ∎

Theorem A.2 A domain element is the *lub* of all the compact elements below it.

Proof Let S be all the compact elements below a domain element d.

From Theorem A.1, $d = \bigcup S$.

Since all elements are deductively closed, $d^\circ = d$, i.e., $(\bigcup S)^\circ = \bigcup S$.

By definition, $\bigsqcup S = (\bigcup S)^\circ$.

Therefore, $d = \bigsqcup S$. ∎

The above theorems show that the compact elements are sufficient to determine all the elements in the domain.

A.2 Domain properties

There are other approaches to defining domains, some stating at the outset the kind of desirable properties that domains should have. For instance, the kinds of domains we need are *consistently complete ω-algebraic cpo's*, and we shall see that information systems do furnish such domains. Some basic definitions:

- A *partial order* (po) is a set with an ordering that is reflexive, antisymmetric and transitive. Clearly, an I.S. domain—with $\subseteq$ as its ordering—is a po.
- A *finitely consistent subset* of a po is one whose every *finite* subset has an ub in the po. A (*strongly*[32]) *consistent subset* of a po is one that (*itself*) has an ub in the po.
- A *directed subset* S of a po is one whose every finite subset has an ub in S. (It follows that directed subsets are also finitely consistent subsets, though not vice versa.)
- A *complete partial order* (cpo) is a po whose every directed subset has a lub. (It follows that directed subsets of a cpo are also (strongly) consistent subsets, though not vice versa.)

Theorem A.3 The set of compact elements below a domain element is directed.

Proof Let $S = \{d_i : d_i \text{ is compact and } d_i \sqsubseteq d\}$.

Let $X = \{\ldots, d_j, \ldots\}$ be a finite subset of S. To show that X has an ub in S.

Since the elements d_j of X are compact, each $d_j = \mathsf{M}_j{}^\circ$ for finite $\mathsf{M}_j = \{\ldots, \mathsf{m}_j, \ldots\}$.

[32]The adjective "strongly" may be dropped.

An ub of X is $e = (\bigcup_j d_j)^\circ$, which is also $(\bigcup_j \mathsf{M}_j)^\circ$, which is a compact element. We need to show that e is in S.

Consider any $\mathsf{p} \in e$. I.e., $\bigcup_j d_j \vdash \mathsf{p}$.

So, $d \vdash \mathsf{p}$, i.e., $\mathsf{p} \in d$.

Thus, the compact element e is a subset of d and therefore approximates it.

Therefore e must be in S. ∎

Theorem A.4 I.S. domains are cpo's.

Proof Let $S = \{\ldots, d_i, \ldots\}$ be a directed subset of the I.S. domain $\mathcal{D}$. To show that S has a lub.

Since S is consistent, we can describe a domain object $s = (\bigcup S)^\circ$.

This is surely an ub for S. To show that it is the least such.

Now suppose e is another ub for S. To show that $s \sqsubseteq e$, i.e., $s \subseteq e$.

Let $\mathsf{p} \in s$. Thus, $\bigcup S \vdash \mathsf{p}$.

Thus, a finite $\mathsf{M} = \{\ldots, \mathsf{m}_j, \ldots\} \vdash \mathsf{p}$, where $\mathsf{M} \subseteq \bigcup S$.

But each m_j belongs to some $d_i \in S$.

And since each $d_i \sqsubseteq e$, we have each $\mathsf{m}_j \in e$.

In other words, $e \vdash \mathsf{p}$.

Since, e is deductively closed, $\mathsf{p} \in e$. Thus, $s \subseteq e$. ∎

Definition A.1 *(Consistent completeness)* A cpo is *consistently complete* if two elements with an ub also have a lub.

Theorem A.5 I.S. domains are consistently complete.

Proof Two domain elements d and e have an ub only if they are consistent. In such an event, $(d \cup e)^\circ$ forms the lub. The proof is similar to the one for Theorem A.4.

∎

Theorem A.6 If d is a compact element and $d \sqsubseteq \bigsqcup S$ for a directed subset S, then $d \sqsubseteq e$ for some e in S.

Proof Let d be a compact element. Thus, $d = \mathsf{M}_d{}^\circ$ for finite $\mathsf{M}_d = \{\ldots, \mathsf{p}_i, \ldots\}$.

Let $S = \{\ldots, s_j, \ldots\}$ be a directed set and let $d \sqsubseteq \bigsqcup S$.

Therefore, $\bigcup S \vdash \mathsf{p}_i$ for each $\mathsf{p}_i \in \mathsf{M}_d$.

I.e., for each i, there is a finite subset $\mathsf{N}_i = \{\ldots, \mathsf{n}_{i,k}, \ldots\}$ of $\bigcup S$ such that $\mathsf{N}_i \vdash \mathsf{p}_i$.

Therefore the set $\bigcup_i \mathsf{N}_i = \{\mathsf{n}_{i,k} : \text{all } i \text{ and } k\}$ will entail *all* the p_i.

But each $\mathsf{n}_{i,k} \in s_{i,k}$ for some $s_{i,k}$.

Thus, $\bigcup_{i,k} s_{i,k}$ entails all the p_i. In other words, $d \subseteq \bigsqcup_{i,k} s_{i,k}$.

But, from the definition of directedness, and noting that the indices i and k have only finite range, $\bigsqcup_{i,k} s_{i,k}$ is an element of S. ∎

Definition A.2 *(Algebraicity)* A cpo is *algebraic* if for any element d, the set $\{d_i : d_i \text{ is compact and } d_i \sqsubseteq d\}$ is directed with d as its lub.

Theorem A.7 I.S. domains are algebraic.

Proof From Theorems A.2 and A.3. ∎

A.3 Domain constructions

Domains can be fashioned out of subdomains, and information systems provide a succinct description of how this can happen. In effect, we determine the information system underlying the result domain from the informations systems of the subdomains. Example constructions include lifted domains, function domains across domains, disjoint unions and cross products. Domain constructions can be *reflexive*, where a recursive equation describes the construction process.

We first look at some basic "primitive" domains. The singleton domain $\mathbf{I} = \{\perp\}$ corresponds to an I.S. with just Δ. The domain $\mathbf{Triv} = \{\perp, \top\}$ corresponds to the I.S. $\{\Delta, \mathsf{p}\}$. The boolean domain $\mathbf{Bool} = \{\perp, tt, ff\}$ corresponds to the I.S. $\{\Delta, tt?, ff?\}$. A more involved example is the domain of natural numbers, $\mathbf{N}_\perp$, which consists of all the (mutually inconsistent) natural numbers above a bottom element. The corresponding I.S. is $\{\Delta, 0?, 1?, \ldots\}$. All the elements of these domains are compact.

Finally, since we describe the constructed domains by way of describing their information systems, it follows that all the domain constructions preserve the properties of consistent completeness and algebraicity.

A.3.1 Lifting

The set of propositions in the I.S. corresponding to $\mathbf{A}_\perp$, the *lift* of $\mathbf{A}$ is:

$$\mathcal{A}_\perp = \{\mathsf{p} : \mathsf{p} \text{ is } \lambda x.x = \mathsf{up}(y) \text{ and } y \text{ contains an } \mathsf{a} \in \mathcal{A}\}; \Delta_{\mathcal{A}_\perp}$$

where $\Delta_{\mathcal{A}_\perp}$ is a new least informative proposition lower than $\Delta_{\mathcal{A}}$.

A more succinct description of the propositions would be:

$$\{\mathsf{up}(\mathsf{a}) : \mathsf{a} \in \mathcal{A}\}; \Delta_{\mathcal{A}_\perp}$$

The domain $\mathbf{A}_\perp$ based on this I.S. has the form:

$$\mathbf{A}_\perp = \{\mathsf{up}(d) : d \in \mathbf{A}\}; \perp_{\mathbf{A}_\perp}$$

with the expected approximation ordering.

A.3.2 Cross product

The set of propositions in the I.S. corresponding to the cross product of $\mathbf{A}$ and $\mathbf{B}$ is:

$$\mathcal{A} \otimes \mathcal{B} = \begin{cases} & \{\mathsf{p} : \mathsf{p} \text{ is } \lambda v.(\mathsf{fst}(v) \subseteq \mathcal{A} \setminus \{\Delta_{\mathcal{A}}\} \text{ and } \mathsf{snd}(v) \subseteq \mathcal{B} \setminus \{\Delta_{\mathcal{B}}\})\} \\ \cup & \{\Delta_{\mathcal{A}\otimes\mathcal{B}}\} \end{cases}$$

where

$$\begin{aligned} \mathsf{fst}((x : \mathbf{A}, y : \mathbf{B})) &= x \\ \mathsf{snd}((x : \mathbf{A}, y : \mathbf{B})) &= y \end{aligned}$$

A more succinct description of the propositions would be:

$$\begin{aligned} & \{(\mathsf{p}, \mathsf{q}) : \mathsf{p} \in \mathcal{A} \setminus \{\Delta_{\mathcal{A}}\} \text{ and } \mathsf{q} \in \mathcal{B} \setminus \{\Delta_{\mathcal{B}}\}\} \\ \cup \; & \{\Delta_{\mathcal{A} \otimes \mathcal{B}}\} \end{aligned}$$

Entailment is as expected: a set of pairs entail (p, q) if the former's first and second elements respectively entail p and q. The new element $\Delta_{\mathcal{A}\otimes\mathcal{B}}$ is entailed by everything.

The domain $\mathbf{A} \otimes \mathbf{B}$ based on this I.S. is:

$$\mathbf{A} \otimes \mathbf{B} = \begin{cases} & \{(d, e) : d \in \mathbf{A} \setminus \{\bot_{\mathbf{A}}\} \text{ and } e \in \mathbf{B} \setminus \{\bot_{\mathbf{B}}\}\} \\ \cup & \{\bot_{\mathbf{A}\otimes\mathbf{B}}\} \quad \text{(new (collapsed) bottom)} \end{cases}$$

It has the expected approximation ordering.

A.3.3 Disjoint union

The set of propositions in the I.S. corresponding to the disjoint union of $\mathbf{A}$ and $\mathbf{B}$ is:

$$\mathcal{A} \oplus \mathcal{B} = \begin{cases} & \{\mathsf{p} : \mathsf{p} \text{ is } \lambda x.(\mathsf{outR}(x) = \varnothing \text{ and } \mathsf{outL}(x) \text{ contains a non-}\Delta_{\mathcal{A}}\ \mathsf{a} \in \mathcal{A})\} \\ \bigcup & \{\mathsf{q} : \mathsf{q} \text{ is } \lambda x.(\mathsf{outL}(x) = \varnothing \text{ and } \mathsf{outR}(x) \text{ contains a non-}\Delta_{\mathcal{B}}\ \mathsf{b} \in \mathcal{B})\} \\ \cup & \{\Delta_{\mathcal{A}\oplus\mathcal{B}}\} \end{cases}$$

A more succinct description of the propositions would be:

$$\begin{aligned} & \{\mathsf{inL}(\mathsf{a}) : \mathsf{a} \in \mathcal{A} \setminus \{\Delta_{\mathcal{A}}\}\} \\ \cup \; & \{\mathsf{inR}(\mathsf{b}) : \mathsf{b} \in \mathcal{B} \setminus \{\Delta_{\mathcal{B}}\}\} \\ \cup \; & \{\Delta_{\mathcal{A}\oplus\mathcal{B}}\} \end{aligned}$$

The operators $\mathsf{outL} : (\mathbf{A} \oplus \mathbf{B}) \to \mathbf{A}$ and $\mathsf{outR} : (\mathbf{A} \oplus \mathbf{B}) \to \mathbf{B}$ are the projections corresponding to the injections $\mathsf{inL} : \mathbf{A} \to (\mathbf{A} \oplus \mathbf{B})$ and $\mathsf{inR} : \mathbf{B} \to (\mathbf{A} \oplus \mathbf{B})$.

Entailment on $\mathcal{A} \oplus \mathcal{B}$ is defined in the obvious way, with the left elements (tagged inL) and right elements (tagged inR) inconsistent, and everything entailing the new element $\Delta_{\mathcal{A}\oplus\mathcal{B}}$.

The domain $\mathbf{A} \oplus \mathbf{B}$ based on this I.S. has the expected form:

$$\mathbf{A} \oplus \mathbf{B} = \begin{cases} & \{\mathsf{inL}(d) : d \in \mathbf{A} \setminus \{\bot_{\mathbf{A}}\}\} & \text{(left subdomain)} \\ \cup & \{\mathsf{inR}(e) : e \in \mathbf{B} \setminus \{\bot_{\mathbf{B}}\}\} & \text{(right subdomain)} \\ \cup & \{\bot_{\mathbf{A}\oplus\mathbf{B}}\} & \text{(new (collapsed) bottom)} \end{cases}$$

and with the expected approximation ordering.

A.3.4 Function domain

Each proposition in the I.S. $\mathcal{A} \to \mathcal{B}$ for the continuous function space $\mathbf{A} \to \mathbf{B}$ is of the form:

$$\lambda f.\text{if finite } \mathsf{M} \subseteq x \in \mathbf{A} \text{ then } \mathsf{p} \in f(x) \in \mathbf{B}$$

which we shall abbreviate as $\langle \mathsf{M}, \mathsf{p} \rangle$. Clearly, M is not only finite but also consistent. $\Delta_{\mathcal{A}\to\mathcal{B}}$ is the tuple $\langle \varnothing, \Delta_{\mathcal{B}} \rangle$.

The propositions $\langle \mathsf{M}_i, \mathsf{p}_i \rangle, \ldots$ are consistent if whenever $\bigcup_j \mathsf{M}_j$ is consistent, then so is $\{\mathsf{p}_j, \ldots\}$.

Entailment is as follows: $\{\ldots, \langle \mathsf{M}_i, \mathsf{n}_i \rangle, \ldots\} \vdash \{\ldots, \langle \mathsf{U}_j, \mathsf{v}_j \rangle, \ldots\}$ if whenever $\mathsf{U}_k \vdash \bigcup\{\ldots, \mathsf{M}_p, \ldots\}$ then the corresponding $\{\ldots, \mathsf{n}_p, \ldots\} \vdash \mathsf{v}_k$.

The I.S. for all the continuous functions includes *all* possible tuples $\langle \mathsf{M}, \mathsf{n} \rangle$. To restrict our I.S. to only the *strict* functions, $\mathcal{A} \to_s \mathcal{B}$, the tuples $\langle \mathsf{M}, \mathsf{n} \rangle$ included must satisfy the restriction: if $\varnothing \vdash \mathsf{M}$ then $\varnothing \vdash \mathsf{n}$.

By this definition, the functions are:

1. *monotonic*: if $a \sqsubseteq b$, then $f(a) \sqsubseteq f(b)$; and

2. *continuous*: if $S = \{\ldots, a_i, \ldots\}$ is a directed subset of $\mathbf{A}$, then $f(S) = \{\ldots, f(a_i), \ldots\}$ is a directed subset of $\mathbf{B}$, and $\bigsqcup f(S) = f(\bigsqcup S)$.

Compact functions are defined as usual. However, from the way we defined our propositions, we can readily identify a smaller collection of *precompact*[33] functions that can determine the whole of the domain. A precompact function can also be viewed as the deductive closure of a single tuple (atomic proposition in the I.S.). The notion can be extended to all domains—not just function domains—by defining a precompact element as the deductive closure of a single atomic proposition in the I.S.

> **Definition A.3** *(Precompact element)* A domain element d is precompact if it has a minimal representation that is a singleton.

In the case of function domains, a precompact element is of the form $\{\mathsf{M}, \mathsf{n}\}^\circ$. In other words, this precompact function maps everything above the compact element $a = \mathsf{M}^\circ$ to the *pre*compact element $b = \{\mathsf{n}\}^\circ$, and everything below to $\perp$. A more succinct representation would be: $a \Rightarrow b$. Every function element (compact or noncompact) can now be represented as the lub of the (directed!) set of precompact functions below it: $\bigsqcup\{\ldots a_i \Rightarrow b_i \ldots\}$, where the a_i are compact and the b_i are precompact.

The form $a \Rightarrow b$, where *both* a and b are compact (i.e., b is not necessarily precompact) is an abbreviation for a special kind of compact procedure. This is the one that is the lub of the precompact procedures $a \Rightarrow b_i$ ($i = 1$ through n), where each b_i is of course precompact. The arrow notation can be extended to forms like $(a \Rightarrow b) \Rightarrow (c \Rightarrow d)$, etc. So long as all the letters denote compact elements, the result denotes a compact element too.

[33]The terms "*prefinite*," "*threshold*," and "*step*" function are also used.

The thing to note is that in any precompact function $a \Rightarrow b$, b is always a precompact element, possibly a precompact function. Writing b out into *its* arrow notation, and continuing the process for *its* argument precompact element, etc., it is always possible to get an expanded form for a precompact function such as $a_0 \Rightarrow \ldots a_i \Rightarrow \ldots a_n$, where the final a_n is an *observable*. The *length* of this function is $n+1$. An advantage of this form is that, for all $j = 0$ through n, the "tails" $a_j \Rightarrow \ldots a_i \Rightarrow \ldots a_n$ are precompact.

A.3.5 Reflexive domain

A reflexive domain satisfies an equation *such as*:

$$\mathbf{D} = \mathbf{O} \oplus (\mathbf{D} \rightarrow \mathbf{D})_{\perp},$$

where $\mathbf{D}$ contains some observable values and all the functions on itself. To construct the information system $\mathcal{D}$, given $\mathcal{O}$, the usual methods for disjoint union and function domain are used, building up the atomic function propositions from scratch, i.e., $\mathcal{O}$:

1. $\Delta_{\mathcal{D}}$ is the (new) least informative member;

2. $\mathsf{inL}(\mathsf{p}) \in \mathcal{D}$ if $\mathsf{p} \in \mathcal{O}$;

3. $inr\langle \mathsf{M}, \mathsf{n} \rangle \in \mathcal{D}$ if M is a finite consistent subset of $\mathcal{D}$ and $\mathsf{n} \in \mathcal{D}$;

4. $inl(\mathsf{p})$ and $inl(\mathsf{q})$ are consistent if p and q are; $inr(\mathsf{p})$ and $inr(\mathsf{q})$ are consistent if p and q are.

Bibliography

[1] H. Abelson and G. J. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Mass., 1985.

[2] A. Appel. *Compiling with Continuations.* Cambridge, 1992.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, revised edition, 1984.

[4] H. P. Barendregt. Functional programming and lambda calculus. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume Vol. B: Formal Models and Semantics, chapter 7, pages 321–363. The MIT Press/Elsevier, 1990.

[5] W. Clinger, J. Rees, et al. Revised^4 Report on the Algorithmic Language Scheme, November 1991.

[6] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[7] O.-J. Dahl and C. A. R. Hoare. Hierarchical program structure. In O.-J. Dahl, E. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming.* Academic Press, 1972.

[8] O. Danvy and A. Filinski. Abstracting control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.

[9] B. F. Duba, R. Harper, and D. MacQueen. Typing first-class continuations in ML. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 163–173, 1991.

[10] R. K. Dybvig and R. Hieb. Engines from Continuations. *Journal of Computer Languages* (Pergamon Press), 14(2):109–124, 1989.

[11] Dylan — an object-oriented dynamic language. Apple Computer Eastern Research and Technology, 1992.

[12] P. Naur (ed.). Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[13] M. Felleisen. Transliterating Prolog into Scheme. Technical Report 182, Indiana University Computer Science Department, 1985.

[14] M. Felleisen. *The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Indiana University, 1987.

[15] M. Felleisen. λ-v-CS: An Extended λ-Calculus for Scheme. In *Proc. 1988 Conference on Lisp and Functional Programming*, pages 72–84, 1988.

[16] M. Felleisen. The Theory and Practice of First-Class Prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 180–190, 1988.

[17] M. Felleisen. The principles of programming languages. Course Notes, COMP 511, Rice University, 1989.

[18] M. Felleisen. On the expressive power of programming languages. In N. Jones, editor, *Proc. 1990 European Symposium on Programming*, pages 134–151. Lecture Notes in Computer Science 432, 1990.

[19] M. Felleisen and R. S. Cartwright. Extended direct semantics. Technical Report 105, Rice University, January 1990.

[20] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987. Preliminary version: Reasoning with Continuations, *Proc. Symposium on Logic in Computer Science*, 1986, 131–141.

[21] A. Filinski. Representing monads. *Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, 1994.

[22] D. P. Friedman, C. T. Haynes, and E. Kohlbecker. Programming with continuations. In P. Pepper, editor, *Program Transformations and Programming Environments*, pages 263–274. Springer-Verlag, Heidelberg, 1985.

[23] R. Harper. Introduction to Standard ML. LFCS Report Series ECS-LFCS-86-14, University of Edinburgh, 1986.

[24] C. T. Haynes. Logic Continuations. *J. Logic Program.*, 4:157–176, 1987. Preliminary version: In *Proc. of the Third International Conference on Logic Programming*, July 1985, London, England, *Lecture Notes in Computer Science*, Vol. 225, Springer-Verlag, Berlin, 671–685.

[25] C. T. Haynes and D. P. Friedman. Abstracting Timed Preemption with Engines. *Journal of Computer Languages* (Pergamon Press), 12(2):109–121, 1987. Preliminary version: *Engines Build Process Abstractions.* In *Proc. Conference on Lisp and Functional Programming*, 1985, 18–24.

[26] C. T. Haynes and D. P. Friedman. Embedding Continuations in Procedural Objects. *ACM Transactions on Programming Languages and Systems*, 9(4):245–254, 1987.

[27] C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining Coroutines from Continuations. *Journal of Computer Languages* (Pergamon Press), 11(3/4):109–121, 1986.

[28] R. Hieb and R. K. Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 128–136, 1990.

[29] R. Hieb, R. K. Dybvig, and C. W. Anderson III. Subcontinuations. *Lisp and Symbolic Computation*, 7:83–109, 1994.

[30] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 2nd edition, 1988.

[31] P. J. Landin. A Correspondence between Algol 60 and Church's Lambda Notation. *Communications of the ACM*, 8(2):89–101; 158–165, 1965.

[32] C. D. Marlin. *Coroutines – A Programming Methodology, a Language Design and an Implementation*, volume 95 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 1980.

[33] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[34] J. McCarthy et al. *Lisp 1.5 Programmer's Manual*. The MIT Press, 2nd edition, 1965.

[35] A. R. Meyer and J. R. Riecke. Continuations may be unreasonable. In *Proc. 1988 Conference on Lisp and Functional Programming*, pages 63–71, 1988.

[36] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[37] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts and London, England, 1990.

[38] J. A. Padget, G. Nuyens, and H. Bretthauer. An overview of EuLisp. *Lisp and Symbolic Computation*, 6(1/2):9–98, 1993.

[39] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[40] C. Queinnec and B. Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 174–184, 1991.

[41] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. ACM Conference*, pages 717–740, 1972.

[42] J. Riecke. Delimiting the scope of effects. *ACM Conference on Functional Programming and Computer Architecture*, 1993.

[43] D. S. Scott. Domains for denotational semantics. In *Proc. International Conference on Automata, Languages, and Programming*, Berlin, 1982. Lecture Notes in Mathematics 140, Springer-Verlag.

[44] K. Sieber. Relating full abstraction results for different programming languages. In K. V. Nori and C. E. Veni Madhavan, editors, *10th Conference on Foundations of Software Engineering and Theoretical Computer Science*, pages 373–387, Berlin, 1990. Springer-Verlag. Lecture Notes in Computer Science 472.

[45] D. Sitaram. Handling control. *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 147–155, 1993.

[46] D. Sitaram and M. Felleisen. Control Delimiters and Their Hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

[47] D. Sitaram and M. Felleisen. Reasoning with Continuations II: How to Get Full Abstraction for Models of Control. In *Proc. 1990 Conference on Lisp and Functional Programming*, pages 161–175, 1990.

[48] D. Sitaram and M. Felleisen. Modeling Continuations without Continuations. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 185–196, 1991.

[49] G. L. Steele Jr. *Common Lisp: the Language.* Digital Press, 2nd edition, 1990. 1st ed., 1984.

[50] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[51] J. E. Stoy and C. Strachey. OS6: An operating system for a small computer. *Comp. J.*, 15(2):117–124, 195–203, 1972.

[52] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Report PRG-11, Oxford University Computing Laboratory, Programming Research Group, 1974.

[53] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

[54] G. J. Sussman and G. L. Steele Jr. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.