Abstract Continuations:
A Mathematical Semantics for Handling Full Functional Jumps

by

Matthias Felleisen, Rice University; Mitchell Wand, Northeastern University;
Daniel P. Friedman and Bruce F. Duba, Indiana University

TECHNICAL REPORT NO. 248

Abstract Continuations: A Mathematical Semantics For Handling Full Functional Jumps

by

M. Felleisen, M. Wand, D. P. Friedman and B. F. Duba

May, 1988

To appear in Proceedings 1988 ACM Symposium on LISP and Functional Programming, July 25–27, 1988.

Abstract Continuations: A Mathematical Semantics for Handling Full Functional Jumps

Matthias Felleisen*
Department of Computer Science, Rice University, Houston, TX 77251-1892

Mitchell Wand†
College of Computer Science, Northeastern University, Boston, MA 02115

Daniel P. Friedman, Bruce F. Duba‡
Computer Science Department, Indiana University, Bloomington, IN 47405

Abstract

Continuation semantics is the traditional mathematical formalism for specifying the semantics of non-local control operations. Modern Lisp-style languages, however, contain advanced control structures like full functional jumps and control delimiters for which continuation semantics is insufficient. We solve this problem by introducing an abstract domain of *rests* of computations with appropriate operations. Beyond being useful for the problem at hand, these *abstract continuations* turn out to have applications in a much broader context, *e.g.*, the explication of parallelism, the modeling of control facilities in parallel languages, and the design of new control structures.

*Partly supported by DARPA/NSF grant CCR 87-20277.
†Partly supported by NSF grant DCR 86-05218
‡Both authors partly supported by NSF grant CCR 87-02117.

1 Continuation Semantics and Functional Jumps

Continuation semantics is the accepted mathematical formalism for specifying languages with non-local control operations. The key idea of the formalism was introduced by Strachey and Wadsworth [17]. In addition to the environment function for assigning meaning to free variables, continuation semantics works with yet another piece of context information: the continuation function for representing the rest of the computation relative to a syntactic phrase.

With continuation semantics, it is possible to specify the denotational semantics of a broad variety of control facilities. Examples are expression exits, **gotos** with label variables, Scheme's call/cc and continuation objects, Reynolds's **escape**, coroutines, and backtracking [11, 15, 17]. Recently, however, a new class of control operations has emerged in Lisp-style programming languages for which traditional continuation semantics is insufficient [1, 3, 5, 12]. These operations roughly correspond to *functional jumps* and *constrained labels*. Unlike a traditional label, a constrained label only represents a part of the rest of a program, namely, that from its occurrence to some control delimiter; similarly, a functional jump does not abandon the current thread of activity, but, instead, invokes the activity associated with the label as if it were an ordinary program function.

¿From an abstract point of view, functional jumps and control delimiters integrate linguistic facilities from operating systems and programming environments into programming languages. For example, running a program as a process totally constrains control actions by the program. Stoy and Strachey [16] were the first to incorporate this kind of action into programming languages in a systematic manner; their exposition of OS6 amply demonstrates the benefits of such a language-based approach to the design of operating systems. In the same vein, the possibility of re-starting partial programs in different stores has the flavor of using these programs as functions. Johnson and Duggan [5] have recently shown how the integration of such programming environment operations into a language simultaneously enhances the language and its environment. Finally, Smith [12] has explored a single-language approach to programming with nested layers of evaluator loops: an integration of the programming environment, operating system, and machine level of a rather different kind.

The difficulty of understanding functional jumps in terms of continuation semantics is due to the choice of representing the rest of a computation as a simple function from intermediate values to final answers. Thus, for example, it is easy to model a traditional label as a continuation and to let a jump denote the replacement of the current continuation with the label-continuation. However, this model does not extend to functional jumps. According to the description of the label-semantics, a functional jump would have to merge the two respective continuations, yet, there is no such operation for the domain of continuation functions. The crucial point is that the representation of rests of computations as simple functions is insufficient to model all possible operations on rests of computations.

Motivated by the above and in keeping with the motto, "if a language can specify something, however odd, the method used to give its formal semantics must be powerful enough to describe it" [17:20], we have generalized continuation semantics to support functional jumps and control delimiters. The result is a new, more abstract view of the domain of continuations and, even more importantly, a generalized method for modeling complex control operations in advanced programming languages, *e.g.*, **gotos** in data-parallel languages like C* [10]. Conversely, the design of abstract continuations also points out that programming language designers have yet to explore a full range of undiscovered control constructs.

In the next section we develop an operational term rewriting semantics for a simple functional language with control operations that directly corresponds to a machine semantics. The third section provides an informal introduction to programming with functional jumps and control delimiters. In the fourth section, after discussing the problems of continuation semantics in more detail, we present our generalization of continuation semantics: an algebra of contexts. In order to show the range of possible implementations and to analyze the relationship to traditional continuation semantics, we explore the initial and final algebra representations in Section 5. The concluding section is a discussion of the potential of abstract continuations for language analysis and design.

2 A Term Rewriting Semantics for Control Operations

The core of our programming language is the prototypical functional language: the untyped λ-value-calculus [8]. Its term set Λ is defined inductively over a set of constants, *Const*, and a set of variables, *Var*:

$$L ::= a \mid x \mid \lambda x.M \mid MN,$$

where a ranges over constants, x over variables, and L, M, and N over Λ-terms. The four classes of expressions have the usual, intuitive semantics: constants stand for basic values and functional primitives, variables are placeholders, λ-abstractions represent call-by-value procedures, and combinations denote function applications.

As usual, the variable x in the abstraction $\lambda x.M$ is called the *bound* variable; a variable is *free* if it does not occur bound. An expression with no free variables is called *closed*. *Programs* in this language are closed expressions. The substitution of a free variable x by N in a term M is denoted by $M[x := N]$.

We specify the semantics of Λ with a term rewriting semantics that evaluates a program to a *value*. The set of values in our system contains constants, variables, and abstractions. This reflects the fact that an ordinary interpreter performs no further evaluation on these entities. For the evaluation of a combination, we first evaluate the function part and then the argument part. Once the two values are available, we perform either a δ-value- or a β-value-rewriting step. As shown by Plotkin [8], this leftmost-outermost rewriting strategy produces a *value* if the program is equivalent to a value in the λ-value-calculus.

Our formalization of a leftmost-outermost rewriting semantics is based on the notion of evaluation contexts. An evaluation context is a term with one hole. The path from the root of an evaluation context to the hole may only pass through applications

and, furthermore, all terms to the immediate left of the path must be values:

$$C[\ ] ::= [\ ] \mid VC[\ ] \mid C[\ ]M.$$

The symbol $[\ ]$ represents the empty context, V ranges over values, M over arbitrary Λ-terms. $C[M]$ represents the term that results from filling the hole in $C[\ ]$ with M.

¿From the above definition it follows that a program is either a value or is uniquely decomposable into an evaluation context and a redex. Furthermore, the redex in the hole of an evaluation context is the leftmost-outermost redex. Hence, we can define rewriting steps for applications by axiom schemata. For constant applications, we have

$$C[fa] \Rightarrow C[\delta(f,a)],$$

where δ is an interpretation on the set of constants

$$\delta\colon \mathit{FuncConst} \times \mathit{BasicConst} \rightarrow \mathit{Const}.$$

For λ-applications, we use the β-value schema:

$$C[(\lambda x.M)V] \Rightarrow C[M[x := V]].$$

That is, a rewriting step transforms a program into another program by replacing the leftmost-outermost δ- or β-value-redex in the evaluation context with its contractum. The evaluation function rewrites a program to a value if there is a chain of rewriting steps from the program to the value:

$$\mathit{eval}(M) = V \text{ iff } M \Rightarrow^* V.$$

In a recent report [2] we have shown that this kind of rewriting semantics is well-suited for specifying and reasoning with non-local control operations. The evaluation context directly represents the rest of the computation relative to the current redex, and thus, it is easy to specify control operations as operations on the evaluation context. Consider an **abort**-operation:

$$L ::= \ldots \mid (\textbf{abort}\ M)\ldots$$

The task of this operation is to terminate a program evaluation and to return the value of M as the final program answer. Within the framework of the rewriting semantics, this is easily specifiable as:

$$C[\textbf{abort}\ M] \Rightarrow M,$$

i.e., **abort** discards the current evaluation context and continues with M.

With a similar rule we can also specify the effect of Reynolds's [9] **escape**-construct

$$L ::= \ldots \mid (\textbf{escape}\ x\ M)\ldots$$

and **escape**-procedures. An **escape**-facility labels the current point in an evaluation by binding x to an **escape**-procedure in M. When invoked, an **escape**-procedure discards its current context and re-installs the encoded context, filling the hole with the argument value. The corresponding rewriting step is:

$$C[\textbf{escape}\ x\ M] \Rightarrow C[M[x := \lambda u.(\textbf{abort}\ C[u])]].$$

It is easy to check that the invocation of an **escape**-procedure has the intended behavior:

$$\begin{aligned} C'(\lambda u.(\textbf{abort}\ C[u]))V &\Rightarrow C'(\textbf{abort}\ C[V]) \\ &\Rightarrow C[V], \end{aligned}$$

that is, the abstraction indeed terminates the current thread of control and returns to the one marked by the corresponding **escape**-construct.

An interesting property of the **escape**-facility is that the new syntactic form almost acts like a null operation, but that it also generates objects, **escape**-procedures, which behave like jumps. This analysis led us to the investigation of a new construct:

$$L ::= \ldots \mid (\textbf{control}\ x\ M)\ldots$$

A **control**-expression transforms the current rest of the computation into a λ-abstraction and binds this abstraction to its variable during the evaluation of the sub-expression:

$$C[\textbf{control}\ x\ M] \Rightarrow M[x := (\lambda u.C[u])].$$

Thus, **control** provides total power over the rest of the computation without introducing another class of objects. If x does not occur free in M, the operation corresponds to an **abort**; on the other hand, if M immediately invokes x, the default continuation for the result of M is the current evaluation context. Hence, **control** is as expressive as **abort** and **escape**.

The interesting point about **control** is that it creates an abstraction from the current control state. When this abstraction is invoked, it can eventually return to the point of invocation. Thus, in

$$\begin{aligned} & 1^+(\textbf{control}\ f\ (f(f0))) \\ \Rightarrow\ & ((\lambda x.1^+x)((\lambda x.1^+x)0)) \\ \Rightarrow\ & ((\lambda x.1^+x)1) \\ \Rightarrow\ & 2 \end{aligned}$$

the composition of f simply effects the duplication of 1^+. This is a concrete example of what we mean by a functional jump: control is transferred to the point of the **control**-construct, but upon reaching the end

of the evaluation context, control returns to the point of invocation.

A problem of all three forms is their unrestricted power over the entire evaluation context. For many applications, it is advantageous to restrict the extent to which **abort**, **escape**, and **control** affect their context. This is one of the motivations for the introduction of a new linguistic facility [1]: the prompt form:

$$L ::= \ldots \mid (\#\ M).$$

Intuitively, the closest #-marker determines the end of the visible evaluation context with respect to a control operation. In order to capture the semantics of #, we must redefine the rewriting rule for **abort**:

$$\begin{array}{ll} & C_1[(\# \cdots (\#\ C_n[\textbf{abort}\ M]) \cdots)] \\ \Rightarrow & C_1[(\# \cdots (\#\ M) \cdots)]; \end{array}$$

the rule for **escape**:

$$\begin{array}{ll} & C_1[(\# \cdots (\#\ C_n[\textbf{escape}\ l\ M]) \cdots)] \\ \Rightarrow & C_1[(\# \cdots \\ & \quad (\#\ C_n[M[l := (\lambda x.(\textbf{abort}\ C_n[x]))]]) \\ & \quad \cdots)]; \end{array}$$

and the rule for **control**:

$$\begin{array}{ll} & C_1[(\# \cdots (\#\ C_n[\textbf{control}\ f\ M]) \cdots)] \\ \Rightarrow & C_1[(\# \cdots (\#\ M[f := (\lambda x.(C_n[x]))]) \cdots)]. \end{array}$$

The notation $C_1[(\# \cdots (\#\ C_n[\cdots]) \cdots)]$ indicates the multiple nesting structure of control delimiters and evaluation contexts. Furthermore, we need a rule to return the final value of a #-form:

$$C_1[(\# \cdots (\#\ V) \cdots)] \Rightarrow C_1[(\# \cdots V \cdots)].$$

An example of a simple use of # is:

$$\begin{array}{ll} & (\#\ (\textbf{control}\ f\ f)(\textbf{control}\ g\ (g(g0))))1^+ \\ \Rightarrow & (\#\ (\lambda x.x(\textbf{control}\ g\ (g(g0)))))1^+ \\ \Rightarrow & (\lambda x.x(\textbf{control}\ g\ (g(g0))))1^+ \\ \Rightarrow & 1^+(\textbf{control}\ g\ (g(g0))) \\ \Rightarrow^+ & 2. \end{array}$$

This illustrates the dynamic character of the #-marker. Even though the **control**-g-expression is lexically inside of the prompt-form, it is unaffected since it is exported from the prompt's scope. More interesting uses of functional jumps and control delimiters are the topic of the next section.

3 Programming with Functional Jumps and Control Delimiters

When embedded in a full-fledged Scheme-like [14, 18] programming language, functional jumps, or *functional continuations*, and control delimiters provide the correct programming paradigm for many situations. For example, many artificial intelligence programs use the so-called agenda-oriented style. The underlying assumption is that a program is a modifiable collection of tasks to which new ones can be added on the fly. In its simplest version, an agenda is a queue with an enqueue-like operation add-last-action for appending yet another task to the program. With functional continuations, this is expressed as:

```
(define add-last-action
  (lambda (task)
    (control l (task (l 'any)))))).
```

Debugging is another area where functional continuations are useful [5, 12]. It is often desirable to define a **break** in a program and to explore the outcome of various resumption-values. With **control**, a **break** is a syntactic abbreviation:

$$(\textbf{break}\ \text{res}) \stackrel{df}{=} (\textbf{control}\ k\ (\textbf{set!}\ \text{res}\ k)),$$

where res is some globally declared identifier. This **break**-construct terminates the program evaluation, saves the control state in an accessible place, and returns to the read-eval-print loop of the interactive system. Given this, it is possible to apply the continuation res to several intermediate values. In an expression like

$$\ldots(\textbf{let}\ [\textit{index}\ (\textbf{break}\ \text{r1})]\ldots)\ldots$$

we can bind *index* to a variety of values and observe the effects. Indeed, since r1 is a functional continuation, we can simply map r1 over a list of values:

```
(map r1 '(1 3 5 7 9)).
```

Depending on whether or not an **abort** should affect the application of r1 or the entire map-expression, we may choose to write

```
(map (lambda (x)  (# (r1 x)))
     '(1 3 5 7 9)).
```

The preceding example can be generalized in a natural way for the intelligent backtracking paradigm. Traditional, Scheme-like continuation objects provide

```
(define print-2-fringes
  (lambda (tree1 tree2)
    (let ([f1 (fringe tree1)] [f2 (fringe tree2)])
      (iterate more ([l1 (f1 'resume)] [l2 (f2 'resume)])
        (cond
          [(and (eof? l1) (eof? l2)) 'done]
          [(eof? l1) (writeln "empty" l2) (f2 'all)]
          [(eof? l2) (writeln l1 "empty") (f1 'all)]
          [else (if (eq? l1 l2) (writeln " " l1 " ") (writeln l1 " " l2))
                (more (f1 'resume) (f2 'resume))])))))

(define fringe
  (lambda (tree)
    (letrec
      ([deliver (lambda (node) (control l (set! LCS l) (contents node)))]
       [LCS (lambda (dummy)
              (iterate L ([tree tree])
                (if (node? tree) (deliver tree) (for-each L tree)))
              'eof)])
      (lambda (msg)
        (case msg
          [resume (# (LCS any))]
          [all (set! deliver writeln) (LCS any) ])))))
```

Figure 1: Print and compare two fringes

---

a basis for implementing such a programming style. We argue, however, that functional continuations are even better suited for this task since it is often necessary to send an intermediate result to a whole list of backtrack continuations. In our framework this becomes:

$$\ldots(\text{map } (\lambda k.(k\ \textit{result}))\ \textit{backtrack-conts})\ldots$$

With Scheme-continuations, on the other hand, this is much harder to realize since the invocation of a continuation eliminates the current thread of control.

Finally, functional continuations offer a better representation for time-sharing processes. Traditionally such processes are represented as continuation objects [19], but this makes it rather difficult to convert a process back into a function. For a concrete, yet simple, example we consider a variant of *Same-fringe*, a well-known problem in the Lisp-community. The original problem is to compare the fringes of two S-expressions without creating intermediate lists. Instead of just comparing two fringes, our variant prints the two fringes such that if the corresponding elements are the same, the element is centered, if they differ, they are separated by a space, and if one of the fringes ends, the other one is printed to the end.

In an extended Scheme, the program has an elegant solution: see Figure 1. The procedure fringe generates a coroutine-like object that returns a node at a time. In addition, the object can convert itself into a function for printing the rest of the fringe by a simple assignment. If one of the fringes ends prematurely, print-2-fringes sends the message 'all to the other coroutine to invoke the printing function. This will finish the printing without further interruptions.

4 Abstract Continuations: An Algebra of Contexts

The continuation semantics of a simple, by-value functional language requires three domains: the do-

main of values, environments, and continuations:

$$\begin{aligned} \rho \in Env &= Var \to Val \\ &\quad \text{(Environments)} \\ m, n, v \in Val &= BasicConst + \\ &\quad [Val \to Cont \to Val] \\ &\quad \text{(Values)} \\ \kappa, \kappa', \ldots \in Cont &= Val \to Val \\ &\quad \text{(Continuations)} \end{aligned}$$

Values are the results of expressions: basic constants represent themselves, functional constants and abstractions (in Λ) yield higher-order functions. Environments map variables to their values. Continuations are functions from values to values that map an intermediate result to a final answer.

The semantic function $\mathcal{E}$ maps Λ-expressions relative to an environment and a continuation to a value:

$$\mathcal{E} : \Lambda \to Env \to Cont \to Val.$$

The defining clauses are:

$$\begin{aligned} \mathcal{E}[\![b]\!] &= \lambda\rho\kappa.\kappa b \\ &\quad \text{for } b \in BasicConst \\ \mathcal{E}[\![f]\!] &= \lambda\rho\kappa.\kappa(\lambda v\kappa'.\kappa'(\delta(f, v))) \\ &\quad \text{for } f \in FuncConst \\ \mathcal{E}[\![x]\!] &= \lambda\rho\kappa.\kappa(\rho[\![x]\!]) \\ \mathcal{E}[\![\lambda x.M]\!] &= \lambda\rho\kappa.\kappa(\lambda v\kappa'.\mathcal{E}[\![M]\!]\rho[x \leftarrow v]\kappa') \\ \mathcal{E}[\![MN]\!] &= \lambda\rho\kappa.\mathcal{E}[\![M]\!]\rho(\lambda m.\mathcal{E}[\![N]\!]\rho(\lambda n.mn\kappa)). \end{aligned}$$

The result of a program M, *i.e.*, a closed expression, is

$$\mathcal{E}[\![M]\!]\rho_{init}(\lambda x.x)$$

where ρ_{init} is an arbitrary environment and $(\lambda x.x)$ is the initial continuation that maps every value to itself.

Equipped with this continuation semantics for the core language, it is straightforward to add defining clauses for **abort**:

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

and **escape**:

$$\mathcal{E}[\![\textbf{escape } x \ M]\!] = \lambda\rho\kappa.\mathcal{E}[\![M]\!]\rho[x \leftarrow \lambda v\kappa'.\kappa v]\kappa.$$

Both clauses reflect the above rewriting steps: **abort** ignores its continuation, **escape** binds its label-variable to an **escape**-procedure. An **escape**-procedure has the functionality of an ordinary value, but upon application it ignores its continuation and passes its argument to the continuation function of the **escape**-construct.

For the semantic equation of the **control**-construct we would like to use a clause similar to the one for **escape**. First, **control** should replace the current continuation with the initial continuation. Second, the corresponding **control**-procedure should not ignore, but should compose its continuation with the continuation of the **control**-point:

$$\begin{aligned} &\mathcal{E}[\![\textbf{control } x \ M]\!] \\ &\quad = \lambda\rho\kappa.\mathcal{E}[\![M]\!] \\ &\qquad \rho[x \leftarrow \lambda v\kappa'.\ \text{“compose } \kappa' \text{ and } \kappa\text{”}\ v] \\ &\qquad (\lambda x.x). \end{aligned}$$

Unfortunately, as mentioned in the introduction, there is no operation on functions for composing two continuations. The “natural” candidate for this combination operation is functional composition: $\kappa' \circ \kappa$. However, this is not sound with respect to the *intended* operational semantics. Consider the program

$$(\lambda x.(\textbf{control } d \ x))(\textbf{control } l \ (1^+(l\,0))).$$

According to *eval*, the expression rewrites to 0; its denotational value according to this proposal would be 1.[1]

The difficulty is that a continuation function maps an *intermediate* value to a *final* result, that is, a continuation function models the *entire* rest of a computation. On the other hand, the requirement to compose two continuations means that at least one of them is *not* representing the entire rest of the computation, but a *partial* rest. Since there is no operation for extracting this partial rest of a computation from a simple function representation, we must conclude that traditional continuation semantics is insufficient for modeling functional jumps.

The solution is a more abstract view of the domain of continuations. What we need is an abstract algebra for modeling the rest of a computation and its operations. Indeed, given the operational understanding of rests of computations as evaluation contexts, it is natural to derive a continuation algebra that models contexts and operations on contexts, in short, *an algebra of contexts*.

For the specification of the algebra of contexts, we follow the traditional approach and define constructors, accessors, and their equational semantics. Let us first recall the definition of a (textual) evaluation

[1] This point is even more obvious when stores are introduced into the semantics and the domain of final answers becomes distinct from the domain of intermediate values.

context:

$$C[\ ] ::= [\ ] \mid VC[\ ] \mid C[\ ]M.$$

An evaluation context is either empty or it is a context in the left or right part of an application. Alternatively, and this view is better for the following semantic analysis, we can perceive a context as being either empty or as being filled with a partial context like $V[\ ]$ (V applied to hole) or $[\ ]M$ (hole applied to M):

$$C[\ ] ::= [\ ] \mid C[V[\ ]] \mid C[[\ ]M].$$

¿From this, it is clear that we need at least two constructors: a 0-ary constructor for the initial context:

$$\Gamma_0 : \rightarrow \mathit{Context};$$

and a 2-ary constructor for filling a context with a partial context:

$$\text{Addframe} : \mathit{Context} \times \mathit{Partial\text{-}context} \rightarrow \mathit{Context}.$$

As we shall see below, the latter constructor indeed covers both actions of filling a context with partial contexts. But, what do partial contexts denote? In the operational semantics, $V[\ ]$, for example, represents the situation when a function is about to be applied to its argument value v in some context κ. If V denotes f, then this means that the partial context denotes the function

$$(\lambda v\kappa.fv\kappa),$$

or, by extensionality, just f. In the traditional continuation semantics this corresponds to the second continuation for an application

$$(\lambda v.fv\kappa),$$

where κ is the continuation of the entire application. By a similar argument, we can derive that the representation for the partial evaluation context $[\ ]M$ is the function:

$$(\lambda f\kappa.\mathcal{E}[\![M]\!]\rho(\text{Addframe}(\kappa, f))).$$

Putting all of this together means that partial contexts are functions ¿from values and contexts to values:

$$\mathit{Partial\text{-}context} = \mathit{Val} \rightarrow \mathit{Context} \rightarrow \mathit{Val}.$$

Since contexts replace continuations in the domain equations, this also implies that values have the same structure as partial contexts:

$$\mathit{Val} = \mathit{Val} \rightarrow \mathit{Context} \rightarrow \mathit{Val}.$$

Following the above analysis of the failure of classical continuation semantics, this is all quite appropriate. Our conclusion was that a context must be an object from which we can extract a *partial* continuation. That is, contexts should consist of intermediate pieces that take a value and a continuation, advance the computation, and, eventually, send off their intermediate result to their continuation.

In order to use contexts, we need an accessor for sending a value to a context. This corresponds to the filling of a context hole with a value. If the context is empty, the evaluation is finished. Otherwise, the partial context around the hole determines the next step. Accordingly, the algebra needs an accessor function

$$\text{Send} : \mathit{Context} \times \mathit{Val} \rightarrow \mathit{Val}.$$

As usual, we define the semantics of an accessor function with equations that determine the meaning of an access relative to a constructor:

$$\begin{aligned} \text{Send}(\Gamma_0, v) &= v \\ \text{Send}(\text{Addframe}(\gamma, f), v) &= fv\gamma. \end{aligned}$$

The two equations precisely formalize the above explanation.

Equipped with this machinery, we can now recast the semantic equations for the core language Λ:

$$\begin{aligned} \mathcal{E}[\![b]\!] &= \lambda\rho\gamma.\text{Send}(\gamma, b) \\ \mathcal{E}[\![f]\!] &= \lambda\rho\gamma.\text{Send}(\gamma, \lambda v\gamma'.\text{Send}(\gamma', \delta(f, v))) \\ \mathcal{E}[\![x]\!] &= \lambda\rho\gamma.\text{Send}(\gamma, \rho[\![x]\!]) \\ \mathcal{E}[\![\lambda x.M]\!] &= \lambda\rho\gamma.\text{Send}(\gamma, \lambda v\gamma'.\mathcal{E}[\![M]\!]\rho[x \leftarrow v]\gamma') \\ \mathcal{E}[\![MN]\!] &= \lambda\rho\gamma.\mathcal{E}[\![M]\!]\rho \\ &\qquad (\text{Addframe}(\gamma, \\ &\qquad\quad (\lambda m\gamma.\mathcal{E}[\![N]\!]\rho \\ &\qquad\qquad (\text{Addframe}(\gamma, m))))). \end{aligned}$$

Next, we may incorporate a clause for **control**. This requires a new accessor

$$\text{Compose} : \mathit{Context} \times \mathit{Context} \rightarrow \mathit{Context}$$

for combining two contexts. In the operational semantics, this corresponds to the invocation of a **control**-procedure where a context $C'[\ ]$ is put into the hole of a context $C[\ ]$:

$$C[(\lambda u.C'[u])V] \Rightarrow C[C'[V]].$$

¿From this, we learn that an empty context adds no information. Thus, the first equation is:

$$\text{Compose}(\gamma, \Gamma_0) = \gamma.$$

Furthermore, when $C[\ ]$ and $C'[\ ]$ are combined, the frames of $C'[\ ]$ become the innermost partial contexts of the resulting context:

$$\begin{aligned}&\text{Compose}(\gamma, \text{Addframe}(\gamma', f))\\ &\quad = \text{Addframe}(\text{Compose}(\gamma, \gamma'), f)\end{aligned}$$

i.e., the frames of the second rest of the computation are used before the frames of the first one. With this, the semantic equation for **control** becomes:

$$\begin{aligned}&\mathcal{E}[\![\textbf{control } x\ M]\!] \qquad (*)\\ &\quad = \lambda\rho\gamma.\mathcal{E}[\![M]\!]\\ &\qquad\qquad \rho[x \leftarrow \lambda v\gamma'.\text{Send}(\text{Compose}(\gamma', \gamma), v)]\\ &\qquad\qquad \Gamma_0.\end{aligned}$$

The addition of **control** illustrates how to deal with a new control facility. For another example, let us consider the #-form, which constrains the visible part of a context. This is easily achieved with a new constructor for marking the context

$$\text{Mark} : \mathit{Context} \rightarrow \mathit{Context},$$

such that the meaning of a prompt becomes

$$\mathcal{E}[\![\#M]\!] = \lambda\rho\gamma.\mathcal{E}[\![M]\!]\rho(\text{Mark}(\gamma)).$$

Two new accessor functions accomplish the separation of the two parts of a context:

$$\begin{aligned}\text{Front} : \mathit{Context} &\rightarrow \mathit{Context}\\ \text{Tail} : \mathit{Context} &\rightarrow \mathit{Context}.\end{aligned}$$

The defining equations are:

$$\begin{aligned}\text{Front}(\Gamma_0) &= \Gamma_0\\ \text{Front}(\text{Addframe}(\gamma, f)) &=\\ &\qquad \text{Addframe}(\text{Front}(\gamma), f)\\ \text{Front}(\text{Mark}(\gamma)) &= \Gamma_0\end{aligned}$$

and

$$\begin{aligned}\text{Tail}(\Gamma_0) &= \Gamma_0\\ \text{Tail}(\text{Addframe}(\gamma, f)) &= \text{Tail}(\gamma)\\ \text{Tail}(\text{Mark}(\gamma)) &= \text{Mark}(\gamma).\end{aligned}$$

The additional equation for Send is:

$$\text{Send}(\text{Mark}(\gamma), v) = \text{Send}(\gamma, v).$$

The accessor Compose is unaffected since its second argument is always an unmarked context.

Given the additional functions for the algebra, we can easily specify the new meaning of **abort**, **control**, and **escape**. An **abort** removes the visible part of the context and determines the meaning of its phrase in the hidden part:

$$\mathcal{E}[\![\textbf{abort } M]\!] = \lambda\rho\gamma.\mathcal{E}[\![M]\!]\rho(\text{Tail}(\gamma)).$$

The **escape**-construct captures the visible front-end and, as before, encodes it as a value:

$$\begin{aligned}&\mathcal{E}[\![\textbf{escape } x\ M]\!]\\ &\quad = \lambda\rho\gamma.\mathcal{E}[\![M]\!]\\ &\qquad\qquad \rho[x \leftarrow \lambda v\gamma'.\text{Send}(\text{Front}(\gamma), v)]\\ &\qquad\qquad \gamma.\end{aligned}$$

A **control**-expression also captures the front-end of the context, but simultaneously removes this part, leaving it to the program whether to invoke or not to invoke the functional continuation:

$$\begin{aligned}&\mathcal{E}[\![\textbf{control } x\ M]\!]\\ &\quad = \lambda\rho\gamma.\mathcal{E}[\![M]\!]\\ &\qquad\qquad \rho[x \leftarrow \lambda v\gamma'.\\ &\qquad\qquad\quad \text{Send}(\text{Compose}(\gamma', \text{Front}(\gamma)), v)]\\ &\qquad\qquad \text{Tail}(\gamma).\end{aligned}$$

The relationship between the *operational* and *denotational* semantics can be expressed with an ordinary adequacy theorem. For Λ with **control** and prompts, we have:

Theorem. *Let M be a program over Λ + **control** + #, b a basic constant, and ρ an arbitrary environment. Then*

$$\mathcal{E}[\![M]\!]\rho\Gamma_0 = b \textit{ iff } \mathit{eval}(M) = b.$$

Proof Sketch. The proof follows the usual line of an adequacy proof. The relationship between evaluation contexts and their denotational counterparts was explained above. From there it follows that if $C[\ ]$ corresponds to γ (relative to ρ), then

$$\mathcal{E}[\![C[M]]\!]\rho\kappa = \mathcal{E}[\![M]\!]\rho(\text{Compose}(\kappa, \gamma))$$

and, hence,

$$\begin{aligned}&\mathcal{E}[\![\lambda x.C[x]]\!]\rho\kappa\\ &\quad = \text{Send}(\kappa, \lambda v\gamma'.\text{Send}(\text{Compose}(\gamma', \gamma), v)).\end{aligned}$$

This is essential for the validation of the equation (*). The rest is routine. □

5 Initial and Final Representations of the Context Algebra

The equational specification of an algebra admits two distinct interpretations and, hence, representations: the initial and the final algebra. The former identifies all algebra elements that must be equal according to the equations, the latter identifies all elements that can be equal without identifying distinct observables.

The initial representation of an algebra is isomorphic to the term algebra [4].[2] For the context algebra with the functions Γ_0, Addframe, Send, and Compose, this leads to a stack-like context structure. More precisely, a context is either empty or a context plus a value. The appropriate domain equation is:

$$\mathit{Context} = \{\Gamma_0\} + [\mathit{Partial\text{-}context} \times \mathit{Context}].$$

If we use $\cdot$ as an infix notation for Addframe, the elements of context typically have the shape

$$v_n \cdot \ldots \cdot v_1 \cdot \Gamma_0,$$

Addframe-operations add an extra element to the left, and Send-operations remove a frame from the left.

Formally, the element Γ_0 is simply a unique data point. The function Addframe is the pairing function:

$$\text{Addframe} = \lambda\gamma f.\langle f, \gamma\rangle.$$

A Send-operation extracts the appropriate pieces:

$$\begin{aligned}\text{Send} = \lambda\gamma v.\ & \textbf{if } is\Gamma_0?(\gamma) \textbf{ then } v \\ & \textbf{else } \text{First}(\gamma) v \text{Second}(\gamma)\end{aligned}$$

Compose, in this framework, becomes an append-operation:

$$\begin{aligned}\text{Compose} = \lambda\gamma\gamma'.\ & \textbf{if } is\Gamma_0?(\gamma') \textbf{ then } \gamma \\ & \textbf{else } \langle \text{First}(\gamma'), \\ & \qquad \text{Compose}(\gamma, \text{Second}(\gamma'))\rangle.\end{aligned}$$

The initial interpretation of contexts supports the intuition that contexts are stacks. This comes as no surprise to implementors, yet this perception is easier to derive from the abstract algebra point of view. Furthermore, this view reveals that functional jumps carry an implementation cost: when Compose is added, we also need an append operation on stacks, which may involve the copying of many stack frames.

[2] This assumes a fixed signature of constructors for partial contexts.

In a Kamin-Wand-style [6, 20] final algebra, elements are represented by tuples whose components are the values of the various accessor functions. Since the two accessor functions of the context algebra, Send and Compose, are parameterized over additional values and contexts, the tuples are pairs of functions:

$$\mathit{Context} = [\mathit{Val} \to \mathit{Val}] \times [\mathit{Context} \to \mathit{Context}].$$

The first component represents Send, the second Compose:

$$\gamma = \langle \lambda v.\text{Send}(\gamma, v), \lambda\gamma'.\text{Compose}(\gamma', \gamma)\rangle.$$

Hence, the accessors are:

$$\begin{aligned}\text{Send} &= \lambda\gamma v.\text{First}(\gamma)v, \\ \text{Compose} &= \lambda\gamma'\gamma.\text{Second}(\gamma)\gamma'.\end{aligned}$$

¿From the definitional clauses it follows that Γ_0 is a pair of two identity functions:

$$\Gamma_0 = \langle \lambda v.v, \lambda c.c\rangle.$$

This guarantees that Send returns the argument value, and that Compose returns the first context.

The definition of Addframe is slightly more complicated:

$$\begin{aligned}&\text{Addframe} \\ &= \lambda\gamma f.\langle \lambda v.fv\gamma, \\ &\qquad \lambda\gamma'.\text{Addframe}((\text{Second}(\gamma)\gamma'), f)\rangle.\end{aligned}$$

When Send operates on a non-empty context, this definition causes the value to be channeled to the right place. The definition of the Compose component reflects the recursive nature of the equational specification. Put abstractly, the decision making is shifted into the tuple-building process, and the components are abstractions of the appropriate right-hand sides of the definitional equations.

The final algebra interpretation closely represents the above view that a context should be an object with components for various tasks. One task is the completion of a computation, another is the partial advancement of a computation. It therefore follows that a restriction to the traditional Send-component yields *ordinary continuation functions.*

6 The Analysis and Design of Control Structures with Abstract Continuations

One result of our development is a denotational semantics for functional jumps and control delimiters.

The more important aspect, however, is a new, widely applicable method for defining the semantics of control facilities. A particularly interesting application concerns the use of parallelism and the inclusion of control facilities in parallel programming languages as in Connection Machine Lisp [13] and C* [10].

Consider the form

$$(\textbf{parallel-or}\ M\ N),$$

which returns true if one of its sub-expressions denotes true, regardless of whether the other expression diverges or not. On one hand, the semantics of this form is directly expressible with the continuous function parallel-or in the value domain of Λ [7]. On the other hand, the use of such a direct semantics as an operational interpreter requires the existence of parallelism on the meta-level. This situation roughly corresponds to the specification of a calling mechanism in a defined language by relying on the calling mechanism on the defining language [9].

An abstract continuation semantics can explicate meta-parallelism. A simple solution is the introduction of two additional context operators: Branch and Switch. Branch combines a context with a thunk—a function waiting for a continuation in order to complete a computation. It indicates that the thunk is an alternative computation with status equal to the current one. This provides the means for specifying the **parallel-or** clause:

$$\begin{aligned}&\mathcal{E}[\![\textbf{parallel-or}\ M\ N]\!]\\ &\quad = \lambda\rho\gamma.\mathcal{E}[\![M]\!]\rho(\mathrm{Branch}(\gamma, \mathcal{E}[\![N]\!]\rho)).\end{aligned}$$

When a branch eventually yields a result, and if the value is true, the continuation is resumed; if the result is false, the alternative branch determines the result of the construct:

$$\begin{aligned}&\mathrm{Send}(\mathrm{Branch}(\gamma, \theta), v)\\ &\quad = \textbf{if}\ \mathrm{True?}(v)\ \textbf{then}\ \mathrm{Send}(\gamma, v)\ \textbf{else}\ \theta\gamma.\end{aligned}$$

When a branch sends off an intermediate value, it is necessary to switch to the alternative computation:

$$\mathrm{Send}(\mathrm{Addframe}(\gamma, f), v) = \mathrm{Switch}(\gamma, (fv)).$$

The accessor Switch extracts the waiting thunk by searching for the closest Branch-frame:

$$\begin{aligned}&\mathrm{Switch}(\mathrm{Addframe}(\gamma, f), \theta)\\ &\quad = \mathrm{Switch}(\gamma, (\lambda\gamma'.\theta\mathrm{Addframe}(\gamma', f)))\end{aligned}$$

and then applies it to an appropriate continuation:[3]

$$\mathrm{Switch}(\mathrm{Branch}(\gamma, \theta_0), \theta) = \theta_0\ \mathrm{Branch}(\gamma, \theta).$$

If there is no Branch, Switch is equivalent to Send:

$$\mathrm{Switch}(\Gamma_0, \theta) = \theta\,\Gamma_0.$$

Given an abstract continuation semantics, it is straightforward to add control operations. Indeed, the introduction of the additional Branch-constructor is entirely orthogonal to the definition of **abort**. Yet, we can also imagine the desirability of a partial abort that cuts a **parallel-or**:

$$\mathcal{E}[\![\textbf{cut}\ M]\!] = \lambda\rho\gamma.\mathcal{E}[\![M]\!]\rho(\mathrm{Cut}(\gamma)).$$

The specification of Cut is similar to Tail, *i.e.*, it pops all frames up to and including the first Branch-frame. In short, we believe that the abstract continuation semantics can specify the meaning of any arbitrary control operation in a deterministic parallel setting.

The second result of our analysis concerns the design of programming languages. Thus far, the control structure of existing languages (and machines) has driven the development of context algebras. The question is whether it makes sense to follow the inverse direction, in other words, whether it is possible to pick an arbitrary continuation domain with some operations and to design a control structure for a language around this domain. Currently, we perceive two different approaches to this question.

First, a programming language could provide *atomic* instead of *global* context operations. That is, instead of offering control operations that affect the entire (visible) context, the programming language could incorporate control facilities that access small parts of a context, *e.g.*, Push, Pop, Top, $is\Gamma_0$? for the typical stack algebra. This has been investigated to some extent in the programming language GL [5]. Other examples remain to be explored.

Second, programming language control structures could be built on top of entirely *different* algebras. One possible algebra could be a queue, which would support the above mentioned agenda control paradigm; another one could be a bag (multi-set), which would more closely correspond to a distributed control system. In short, the context domain could really be an abstraction of the real world system that is to be modeled.

In summary, we believe that there is an entire world of unexplored control paradigms between the well-known sequential and the (deterministic) parallel structures. Abstract continuation algebras provide a means for their exploration.

[3] This definition, although "natural" in some sense, is unfair since branches of nested **parallel-ors** can be starved. The following, minor change can fix this problem:

$$\mathrm{Switch}(\mathrm{Branch}(\gamma, \theta_0), \theta) = \mathrm{Switch}(\gamma, \lambda\gamma'.\theta_0\ \mathrm{Branch}(\gamma', \theta)).$$

References

1. FELLEISEN, M. The theory and practice of first-class prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, 180–190.

2. FELLEISEN, M. AND D.P. FRIEDMAN. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, edited by M. Wirsing. Elsevier Science Publishers B.V. (North-Holland) , Amsterdam, 1986, 193–217.

3. FELLEISEN, M., D.P. FRIEDMAN, B. DUBA, AND J. MERRILL. Beyond continuations. Technical Report No 216, Indiana University Computer Science Department, 1987.

4. GOGUEN, J., J. THATCHER, AND E. WAGNER. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In *Current Trends in Programming Methodology* IV, edited by R. Yeh. Prentice-Hal l, 1979, 80–149.

5. JOHNSON, G.F. AND D. DUGGAN. Stores and partial continuations as first-class objects in a language and its environment. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, 158–168.

6. KAMIN, S. Final data type specifications: A new data type specification method. In *Proc. 7th Symposium on Principles of Programming Languages*, 1980, 131–138.

7. PLOTKIN, G.D. LCF considered as a programming language. *Theor. Comput. Sci.* **5**, 1977, 223–255.

8. PLOTKIN, G.D. Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.* **1**, 1975, 125–159.

9. REYNOLDS, J.C. Definitional interpreters for higher-order programming languages. In *Proc. ACM Annual Conference*, 1972, 717–740.

10. ROSE, J.R. AND G.L. STEELE JR. C*: An extended C language for data parallel programming. Technical Report No. PL87-5, Thinking Machines Corporation, 1987.

11. SCHMIDT, D.A. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Newton, Mass., 1986.

12. SMITH, B.C. Reflection and Semantics in Lisp. In *Proc. 11th ACM Symposium on Principles of Programming Languages*, 1984, 23–35.

14. STEELE, G.L. JR AND G.J. SUSSMAN. The revised report on Scheme, a dialect of Lisp. Memo 452, MIT AI-Lab, 1978.

13. STEELE, G.L. JR AND W.D. HILLIS. Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing. *Proc. 1986 ACM Symposium on Lisp and Functional Programming*, 1986, 279–297.

15. STOY, J.E. *Denotational Semantics: The Scott-Strachey Approach to Programming Languages.* MIT Press, Cambridge, Mass., 1981.

16. STOY, J.E., AND C. STRACHEY. OS6: An operating system for a small computer. *Comp. J.* **15**(2), 1972, 117–124; 195–203.

17. STRACHEY, C. AND C.P. WADSWORTH. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, 1974.

18. SUSSMAN, G.J. AND G.L. STEELE JR. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.

19. WAND, M. Continuation-based multiprocessing. In *Proc. 1980 ACM Conference on Lisp and Functional Programming*, 1980, 19–28.

20. WAND, M. Final algebraic semantics and data type extensions. *J. Comput. Syst. Sci.* **19**, 1979, 27–44.