Continuations in Programming Practice: Introduction and Survey

Matthias Felleisen
Dept. of Computer Science
Rice University
matthias@cs.rice.edu

Amr Sabry
Dept. of Computer and Information Science
University of Oregon
sabry@cs.uoregon.edu

August 26, 1999

Contents

1 Plan

Continuations appear in many contexts including compilers [3, 43, 56, 63], denotational semantics [65, 66], operating systems [14], and classical logic [32, 49, 50, 51]. We can however understand much of the history [55], development, and applications of continuations by simply studying programming practice.

We begin our study of continuations (Section 2) by making an informal connection with control stacks commonly used for runtime storage of activation records. Next (Section 3) we generalize this argument: instead of looking at the evaluation of single programs, we look at entire programming languages and show how continuations appear at the implementation level. Once continuations are revealed as entities in an implementation it becomes natural to give them an explicit representation in the evaluator, and then to manipulate them to express (Section 4), and even discover (Section 5), control operators. The manipulation of continuations using control operators provides an expressive formalism in which one can express elegant solutions to a wide variety of problems (Section 6). Finally we conclude with a perspective about the future uses of continuations (Section 7).

2 Control Stacks and Continuations

In undergraduate compiler courses, computer science students learn that in many common programming languages, one can use a *control stack* to keep track of live procedure activations [2:p.393]. It turns out that control stacks are intimately related to continuations, and hence constitute a good starting point for our investigation.

2.1 Function Calls and Returns

At the machine level, a function call corresponds to the following sequences of actions:

1. save enough information to be able to resume execution of the program when the function returns. This information typically includes the address of the current instruction, the contents of some of the registers, and is saved in a frame on the control stack. Then,

2. *jump* to the code for the function passing the arguments, and

3. restore the saved context passing the return value of the function.

The aggregate information that is needed to continute the execution of the program after the function call returns is the *continuation.* More abstractly, the continuation of a function call is an object which, given the return value of the function, yields the final answer of the program.

A simple example should help. Consider the following C program:

```
int f () { return 1; }

void main () {
```

```
  if (f()) printf("Good f\n");
  else printf("Bad f\n");
}
```

The continuation of the call `f()` is an object, which expects a value `v` and then uses `v` to continue execution till the end of the program. This object can be represented in many ways but a particularly appealing representation is the following function:

```
void f_continuation (int v) {
  if (v) printf("Good f\n");
  else printf("Bad f\n");
}
```

Using this functional representation of the continuation, the original program can be restructured as follows:

```
void f_continuation (int v) {
  if (v) printf("Good f\n");
  else printf("Bad f\n");
}

int f () { return 1; }

void main () {
  f_continuation(f());
}
```

The new structure clarifies and helps optimize the sequence of actions associated with the call `f()`. Given the new structure, this sequence is:

1. save the information needed to resume execution after `f` returns: this information is nothing but the code for `f_continuation`. Then,
2. *jump* to the code for the function `f` passing the arguments, and
3. restore the saved context passing the return value of the function. Since the saved context is nothing but the function `f_continuation`, this action constitutes in calling `f_continuation` passing the return value as an argument.

In other words, the original program is equivalent to:

```
void f_continuation (int v) {
  if (v) printf("Good f\n");
  else printf("Bad f\n");
}

void f () { f_continuation(1); }

void main () { f(); }
```

where we have moved the call to `f_continuation` from around `f()` to around the return value for `f`. Indeed, there is no reason to execute the return and then call `f_continuation`; we might as well call it directly. Furthermore, by managing the continuation explicitly, the call `f()` is reduced to a *tail-call* [64]: there is no need to save anything before the call or restore anything after the call; it is just a jump. This optimization of tail-calls to jumps is called *tail-call optimization* [10].

2.2 Recursive Calls

To gain more insight into the nature of continuations, let's consider a more complicated example, involving a recursive computation:

```
int factorial (int n) {
  if (n == 0) return 1;
  else {
    int v = factorial(n-1);
    return n*v;
  }
}

void main () {
  printf ("Factorial of 5 = %d\n", factorial(5));
}
```

We have named the result of the recursive call to `factorial` to simplify the following discussion. As before, the continuation of the call `factorial(n-1)` is an object which expects a value `v` and then uses `v` to continue execution till the end of the program. However unlike the previous example, the remainder of the program is not syntactically apparent: it includes all pending recursive calls as well as the continuation of the original call to the `factorial` function.

As an example, the continuation of the call `factorial(3)` is an object which expects the value `v` of that call and then completes the execution of the program as follows: multiply `v` by 4, return that result to a pending recursive invocation which multiplies the result by 5 yielding the final value, which is returned to the original call of `factorial` and printed.

That continuation can be represented as a function that takes a value `v`, multiplies by 20, and prints it.

Similarly, the continuation of the call `factorial(2)` is a function which multiplies its argument by 60 and prints the result; the continuation of the call `factorial(1)` is a function which multiplies its argument by 120 and prints the result; and the continuation of the call `factorial(0)` is a function which multiplies its argument by 120 and prints the result.

Thus, in our example, every recursive call `factorial(i-1)` is associated with a continuation that multiplies its argument by `5*4*...*i` and prints the result. We can make this association more explicit by rewriting the `factorial` function so that every call is explicitly passed the relevant part of its continuation:

```
int factorial (int i, int prod) {
  if (i == 0) return prod;
  else return factorial(i-1, prod*i);
}
```

In this version, every call to `factorial` is associated with a number which represents the important information in its continuation. More importantly, the call to `factorial` is now a tail-call and can be optimized as a jump (with arguments):

```
int factorial (int i, int prod) {
 L: {
   if (i == 0) return prod;
   else {
     prod = prod * i; i = i - 1; goto L;
   }
 }
}
```

This final program is the familiar iterative implementation of `factorial` (using explicit jumps rather than a looping construct though). In light of our derivation, this implementation is nothing but a program that explicitly manipulates its continuation by passing it as an extra argument. Programs in this style are said to be in *continuation-passing style* (CPS).

Remarkably, it is possible to automatically convert any program to CPS. The standard transformation [13, 27, 52, 57, 59] represents all continuations as functions, and is only appropriate for languages with higher-order functions like Scheme [9, 42, 1] or SML [47, 48]. For other languages, continuations can be represented as objects or records [30:ch.9]. For specific programs, like `factorial` above, one can sometimes find highly optimized representations of the continuation as numbers or other simple data structures.

Before concluding this section, we invite the reader to follow the development in this section to derive an iterative solution to some naturally recursive problems like the preorder traversal of a binary tree or the "Towers of Hanoi."

3 The Control State of Evaluators

The intuitive correspondence between continuations and control stacks suggests that continuations are present behind the scene in implementations of programming languages. To make this observation precise and to account for implementations that do not use a control stack, we need to study the implementation of a small programming language at an abstract level.

Naturally, the specification of an implementation can be done using a large number of techniques. Some of the most popular ones are: natural semantics [41], rewriting semantics [52], and abstract machines [44, 45, 54]. Not surprisingly, continuations do not appear in the same manner in each of these specification techniques, and hence it will prove illuminating to study each technique in some detail.

Natural semantics does not provide an explicit representation of the continuation and will prove, at best, awkward to use in further developments. The other two specifications, based on term rewriting [18, 20, 23, 24, 67, 69] and abstract machines [17, 22], constitute a major development in the area of continuations: they provide an explicit representation of the continuation and an abstract view of its role in evaluation. They will lead to further insights about continuations.

3.1 Tiny Scheme: Definitions

The language TINY SCHEME is a small subset of a typical higher-order functional language. The expressions of the language are divided into two syntactic categories: values and non-values. Values include constants (integers), variables, and (call-by-value) functions. Non-values include let-expressions (blocks), conditionals, and applications of user-defined functions and operators.

$$\begin{array}{rcll} M,N,L & ::= & V \mid \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 \mid \mathsf{if}\ M\ N\ L \mid M\ N \mid op\ M & (\textit{Terms}) \\ V & ::= & n \mid x \mid \mathsf{fun}\ x.M & (\textit{Values}) \\ op & ::= & \mathsf{add1} \mid \mathsf{sub1} \mid \ldots & (\textit{Operators}) \\ x & \in & \textit{Variables} \quad \text{(An infinite set of variables)} & \\ n & \in & \mathbb{N} \quad \text{(The integers)} & \end{array}$$

In a function ($\mathsf{fun}\ x.M$) or expression ($\mathsf{let}\ x = N\ \mathsf{in}\ M$), the scope of the variable x is the expression M. We say that x is *bound* in M. A variable that is not bound is *free*. As usual, bound variables can be renamed [4]. Thus: ($\mathsf{fun}\ x.(x\ z)) = (\mathsf{fun}\ u.(u\ z)$). The term $M[x := N]$ is the result of the capture-free substitution of all free occurrences of x in M by N. For example, $(\mathsf{fun}\ x.(x\ z))[z := (\mathsf{fun}\ y.x)] = (\mathsf{fun}\ u.(u\ z))[z := (\mathsf{fun}\ y.x)] = (\mathsf{fun}\ u.(u\ (\mathsf{fun}\ y.x)))$.

The formal semantics of TINY SCHEME is a (partial) function *eval* from programs to answers. A *program* is a term with no free variables and an *answer* is either a number or

$$\frac{}{V \Downarrow V} \qquad \frac{M_1 \Downarrow V_1 \qquad M_2[x := V_1] \Downarrow V}{\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 \Downarrow V}$$

$$\frac{M_1 \Downarrow 0 \qquad M_2 \Downarrow V}{\mathsf{if}\ M_1\ M_2\ M_3 \Downarrow V} \qquad \frac{M_1 \Downarrow V_1 \qquad M_3 \Downarrow V}{\mathsf{if}\ M_1\ M_2\ M_3 \Downarrow V}\ V_1 \neq 0$$

$$\frac{M_1 \Downarrow \mathsf{fun}\ x.M \qquad M_2 \Downarrow V_2 \qquad M[x := V_2] \Downarrow V}{M_1\ M_2 \Downarrow V}$$

$$\frac{M \Downarrow 0}{\mathsf{add1}\ M \Downarrow 1} \quad \cdots \quad \frac{M \Downarrow 41}{\mathsf{add1}\ M \Downarrow 42} \quad \cdots$$

$$\frac{M \Downarrow 1}{\mathsf{sub1}\ M \Downarrow 0} \quad \cdots \quad \frac{M \Downarrow 42}{\mathsf{sub1}\ M \Downarrow 41} \quad \cdots$$

Figure 1: Natural Semantics for Tiny Scheme

the special tag `procedure`. For example, we have:

$$\begin{array}{rcl}
eval(7) & = & 7 \\
eval(\mathsf{add1}\ 4) & = & 5 \\
eval(\mathsf{if}\ 0\ 2\ 3) & = & 2 \\
eval(\mathsf{if}\ 1\ 2\ 3) & = & 3 \\
eval(\mathsf{let}\ x = 4\ \mathsf{in}\ (\mathsf{add1}\ x)) & = & 5 \\
eval(((\mathsf{fun}\ x.(\mathsf{add1}\ x))\ 4)) & = & 5 \\
eval(\mathsf{fun}\ x.x) & = & \texttt{procedure} \\
eval(\mathsf{fun}\ x.(\mathsf{add1}\ x)) & = & \texttt{procedure} \\
eval(\mathsf{fun}\ x.(x\ x)) & = & \texttt{procedure} \\
eval(\mathsf{sub1}\ 0) & \text{is} & \text{undefined} \\
eval(((\mathsf{fun}\ x.(x\ x))\ (\mathsf{fun}\ x.(x\ x)))) & \text{is} & \text{undefined}
\end{array}$$

The following subsections will show three different specifications of the function *eval*.

3.2 Natural Semantics

This subsection illustrates one possible specification of the function *eval*, using natural semantics. The rules in Figure 1 are axioms and inference rules of a simple evaluation logic. Each rule specifies how the value of an expression is calculated using the values of its subexpressions.

Using the rules, the evaluation tree of the program,

$$\mathsf{let}\ x = (\mathsf{add1}\ 7)\ \mathsf{in}\ (\mathsf{sub1}\ x)$$

is:

$$\frac{\dfrac{\dfrac{}{7 \Downarrow 7}}{\mathsf{add1}\ 7 \Downarrow 8} \qquad \dfrac{\dfrac{}{8 \Downarrow 8}}{\mathsf{sub1}\ 8 \Downarrow 7}}{\mathsf{let}\ x = (\mathsf{add1}\ 7)\ \mathsf{in}\ (\mathsf{sub1}\ x) \Downarrow 7}$$

The construction of the tree proceeds as follows. To evaluate the let-expression, we must first evaluate the value of the expression (add1 7). The evaluation of this primitive application requires that the argument 7 be first evaluated. Since 7 is already a value, it requires no further evaluation. Once the value of (add1 7) is available, we substitute it for all free occurrences of x in the body (sub1 x) of the let-expression, and evaluate the resulting expression. The rest of the evaluation tree proceeds as expected to compute the final answer 7.

As we construct the tree, we can stop at any point and consider what we need to know/do to finish the evaluation. For example, when evaluating the subexpression (add1 7), the parts of the tree that are needed to produce the final result constitute the continuation. This partial tree can be informally depicted as:

$$\frac{\Downarrow v \qquad \dfrac{\vdots}{(\mathsf{sub1}\ v) \Downarrow}}{\mathsf{let}\ x = (\mathsf{add1}\ 7)\ \mathsf{in}\ (\mathsf{sub1}\ x) \Downarrow}$$

We can think of this partial tree as a function which, given a value for v produces the final answer of the program. This is a rather complicated representation of the continuation and in general, continuations are hard to pinpoint and manipulate in this formulation of the evaluator.

3.3 Term Rewriting

The next specification implements the function *eval* by manipulating the program text. Initially, the state of the evaluator is the program to evaluate. At each step, a subexpression is reduced, yielding a new program. Evaluation terminates when reduction reaches a value.

Definition 3.1 *Let M be a* TINY SCHEME *program, and let $\longmapsto$ be the partial function used to advance the evaluation of programs by one step, and let $\longmapsto^*$ be the reflexive transitive closure of $\longmapsto$ (i.e., the result of applying $\longmapsto$ zero or more times), then:*

- $eval(M) = n$ *if* $M \longmapsto^* n$,
- $eval(M) =$ `procedure` *if* $M \longmapsto^* (\mathsf{fun}\ x.N)$.
- $eval(M)$ *is undefined otherwise.*

Before giving the complete definition of the abstract machine, consider the evaluation of the program,

$$\mathsf{sub1}\ (\mathsf{if}\ 0\ (\mathsf{add1}\ 7)\ 3),$$

which proceeds as follows:

$$\mathsf{sub1}\ (\mathsf{if}\ 0\ (\mathsf{add1}\ 7)\ 3) \longmapsto \mathsf{sub1}\ (\mathsf{add1}\ 7) \longmapsto \mathsf{sub1}\ 8 \longmapsto 7.$$

The first step identifies the subexpression (if 0 (add1 7) 3) as the current instruction, and the context (sub1 [ ]) as the current continuation. The current instruction is then evaluated to 8:

$$(\mathsf{if}\ 0\ (\mathsf{add1}\ 7)\ 3) \longmapsto \mathsf{add1}\ 7 \longmapsto 8$$

This evaluation occurs in the context of the current continuation (sub1 ([ ])). When this continuation receives the subresult 8, the evaluation proceeds to completion.

The process of decomposing a term into a current instruction and a current continuation can be formalized using the notion of *evaluation contexts* [22]:

$$\begin{array}{rcl} E & ::= & [\,] \\ & | & \mathsf{let}\ x = E\ \mathsf{in}\ M \\ & | & \mathsf{if}\ E\ N\ L \\ & | & (E\ N) \mid (V\ E) \mid (op\ E) \end{array}$$

The inductive definition of evaluation contexts shows, for each kind of non-value expression, the location of the current instruction. For example, in an expression of the form ($\mathsf{let}\ x = N\ \mathsf{in}\ M$), the current instruction must occur within N, hence the definition of evaluation contexts includes the clause ($\mathsf{let}\ x = E\ \mathsf{in}\ M$). In an expression of the form ($\mathsf{if}\ M\ N\ L$), the current instruction must occur within M, hence the definition of evaluation contexts includes the clause ($\mathsf{if}\ E\ N\ L$). For expressions of the form $(M\ N)$, the current instruction could be within M or N. We choose to proceed left-to-right and select the current instruction from within M, *i.e.*, we define $(E\ N)$ to be an evaluation context. Only when the term in the function position is a value or an operator, and hence does not contain a current instruction to execute, do we examine the term in the argument position: $(V\ E)$ and $(op\ E)$. Finally, the entire term under consideration could itself be the current instruction, hence we include the empty context [ ] as a base case for the definition.

It is easy to check that every program is either a value (that needs no further evaluation) or can be decomposed into an evaluation context E and an instruction R. Decompositions are transformed according to the following rewriting rules:

$$\begin{array}{rcllr} E[(\mathsf{let}\ x = V\ \mathsf{in}\ M)] & \longmapsto & E[M[x := V]] & & (\beta_v) \\ E[(\mathsf{if}\ 0\ M\ N)] & \longmapsto & E[M] & & (\mathsf{if}_t) \\ E[(\mathsf{if}\ V\ M\ N)] & \longmapsto & E[N] & V \neq 0 & (\mathsf{if}_f) \\ E[((\mathsf{fun}\ x.M)\ V)] & \longmapsto & E[M[x := V]] & & (\mathsf{let}_v) \\ E[(\mathsf{add1}\ n)] & \longmapsto & E[m] & m = n + 1 & (\mathsf{add1}) \\ E[(\mathsf{sub1}\ n)] & \longmapsto & E[m] & n > 0, m = n - 1 & (\mathsf{sub1}) \end{array}$$

As the previous discussion suggests, in this presentation of the evaluator, the continuation of any instruction is represented by the evaluation context E surrounding the instruction. This precise and explicit representation of the continuation will prove helpful in further studies.

3.4 Exercise: Evaluating Other Languages

The representation of continuations using evaluation contexts is a robust notion in the sense that it may be easily adapted to other languages.

Call-by-Name First we outline the changes needed to accommodate a variant of TINY SCHEME with a call-by-name semantics. In a call-by-name language, the arguments to procedures are not evaluated and hence $(V\ E)$ and ($\mathsf{let}\ x = E\ \mathsf{in}\ M$) are no longer evaluation contexts. No other changes are required. To summarize, the set of evaluation contexts in a call-by-name variant of TINY SCHEME is:

$$E \ ::= \ [\,] \mid (\mathsf{if}\ E\ M\ N) \mid (E\ M) \mid (op\ E)$$

We invite the reader to complete the development by specifying the call-by-name evaluation function $eval_n$ as a set of rewriting rules.

An Imperative Language As a more complicated exercise, we adapt the development to the following simple imperative language:

$$\begin{array}{rcll}
W & ::= & \mathsf{skip} \ \mid \ x = P \ \mid \ W ; W \ \mid \ \mathsf{while}\ P\ \mathsf{do}\ W \mid \mathsf{if}\ P\ \mathsf{then}\ W_1\ \mathsf{else}\ W_2 & \\
P & ::= & n \ \mid \ x \ \mid \ P + P \ \mid \ P - P & \\
x & \in & \mathit{Variables} & \text{(An infinite set of variables)} \\
n & \in & \mathbb{N} & \text{(The integers)}
\end{array}$$

A program in this language is a statement W. An empty statement is denoted by skip. Statements include assignments which also serve as declarations. A sequence of statements separated by ; is itself a statement. Both while- and if-statements treat 0 as the true value. Expressions only appear in the following restricted contexts: the right-hand-side of an assignment, and the test part of while- and if-statements. Expressions include variables and integers, and can be combined using addition and subtraction. For example, the following program multiplies the values of x and y and stores the result in r:

$$\begin{array}{l}
x = 5; \\
y = 8; \\
r = 0; \\
\mathsf{if}\ x\ \mathsf{then}\ z = 1\ \mathsf{else}\ z = 0; \\
\mathsf{while}\ z\ \mathsf{do} \\
\qquad x = x - 1; \\
\qquad r = r + y; \\
\qquad \mathsf{if}\ x\ \mathsf{then}\ z = 1\ \mathsf{else}\ z = 0
\end{array}$$

As in the previous section, evaluation using a term rewriting machine, consists of decomposing the current program into a current continuation and a current instruction, and performing the current instruction. As an example, consider the following program:

$$\begin{array}{l}
x = (1 + 2) + 3; \\
y = x + 4
\end{array}$$

The current instruction is $(1 + 2)$ and the continuation is:

$$\begin{array}{l}
x = [\,] + 3; \\
y = x + 4
\end{array}$$

It is convenient to split the continuation in two parts: an expression continuation $([\,] + 3)$, and a command continuation $[\,] \,; y = x + 4$.

As for TINY SCHEME, we use evaluation contexts to represent continuations. The definition of expression continuations specify a left-to-right evaluation order of arithmetic expressions:

$$E \quad ::= \quad [\,] \mid E+P \mid n+E \mid E-P \mid n-E$$

Command continuations are similarly defined using another family of contexts:

$$C \quad ::= \quad [\,] \mid C;W$$

Intuitively, the current command is the first command is a sequence; the remainder of the sequence forms the continuation.

Before we can specify the rewriting rules, we must adapt the formalism of rewriting to the context of imperative languages. In the functional language TINY SCHEME, each variable was substituted with its value, and there was no need to build any data structure to maintain the values of variables. In the imperative language, we cannot substitute variables with their values as the following example shows:

$$\begin{array}{l} z = 1; \\ \textsf{while}\ z\ \textsf{do} \\ \qquad z = z - 1 \end{array}$$

If we substitute the first occurrence of z by the value 1, we get:

$$\begin{array}{l} z = 1; \\ \textsf{while}\ 1\ \textsf{do} \\ \qquad z = z - 1 \end{array}$$

which would change a terminating program to an infinite loop.

Instead of substitution, we need to maintain a data structure representing the heap that associates every variable with its current value. Assignments will update this data structure, and uses of variables will lookup the current value in the data structure. Fortunately, there is a simple term representation for this data structure as a sequence of assignments [6, 46, 58]:

$$S \quad ::= \qquad x_1 = n_1; \ldots; x_k = n_k$$

If $x = n$ is in the list S, we say that $S(x) = n$. We assume in this simple language that all variables are distinct and hence there are no conflicts when looking in the heap.

Every decomposition of a program will consist of a command continuation C filled with the current heap and the current statement. The current statement may be further decomposed using expression continuations to reveal the next subexpression to evaluate:

$$
\begin{array}{rcll}
C[S; x = E[y]] & \longmapsto & C[S; x = E[n]] & S(y) = n \\
C[S; x = E[n_1 + n_2]] & \longmapsto & C[S; x = E[n]] & n = n_1 + n_2 \\
C[S; x = E[n_1 - n_2]] & \longmapsto & C[S; x = E[n]] & n = n_1 - n_2
\end{array}
$$

$$
C[S; \mathsf{while}\ P\ \mathsf{do}\ W] \longmapsto C[S; \mathsf{if}\ P\ \mathsf{then}\ (W; \mathsf{while}\ P\ \mathsf{do}\ W)\ \mathsf{else\ skip}]
$$

$$
\begin{array}{rcll}
C[S; \mathsf{if}\ 0\ \mathsf{then}\ W_1\ \mathsf{else}\ W_2] & \longmapsto & C[S; W_1] & \\
C[S; \mathsf{if}\ n\ \mathsf{then}\ W_1\ \mathsf{else}\ W_2] & \longmapsto & C[S; W_2] & n \neq 0 \\
C[S; \mathsf{if}\ E[y]\ \mathsf{then}\ W_1\ \mathsf{else}\ W_2] & \longmapsto & C[S; \mathsf{if}\ E[n]\ \mathsf{then}\ W_1\ \mathsf{else}\ W_2] & S(y) = n \\
C[S; \mathsf{if}\ E[n_1 + n_2]\ \mathsf{then}\ W_1\ \mathsf{else}\ W_2] & \longmapsto & C[S; \mathsf{if}\ E[n]\ \mathsf{then}\ W_1\ \mathsf{else}\ W_2] & n = n_1 + n_2 \\
C[S; \mathsf{if}\ E[n_1 - n_2]\ \mathsf{then}\ W_1\ \mathsf{else}\ W_2] & \longmapsto & C[S; \mathsf{if}\ E[n]\ \mathsf{then}\ W_1\ \mathsf{else}\ W_2] & n = n_1 - n_2
\end{array}
$$

$$
C[S; \mathsf{skip}; W] \longmapsto C[S; W]
$$

Again, a precise and explicit representation of the continuation using expression and command contexts would prove helpful if we were to study extensions of this imperative language with advanced control constructs.

3.5 The CEK Machine

The CEK machine [22] is another possible specification of the evaluation function for Tiny Scheme. The machine has three components: an expression, an environment, and a continuation. The expression C refers to the current expression of interest to the evaluator (initially the entire program). The environment ρ[1] is a data structure that includes bindings for the free variables in C. The continuation K is the machine's representation of the evaluation context. The use of environments instead of substitution gives a more explicit representation of the continuation.

During evaluation, the machine proceeds through a sequence of configurations or states. We first present an example, and then give the general definition. To evaluate the term sub1 (if 0 (add1 7) 3), we initialize the machine as follows:

$$
\langle \mathsf{sub1}\ (\mathsf{if}\ 0\ (\mathsf{add1}\ 7)\ 3), \emptyset, \underline{\mathsf{stop}} \rangle
$$

Initially, the environment register holds no bindings, and the continuation register holds <u>stop</u> which indicates the rest of the computation at this point. Evaluation proceeds as

[1] To avoid confusing environments with evaluation contexts, we will refer to environments with the greek letter ρ and reserve E for evaluation contexts.

follows:

$$\begin{array}{rl}
& \langle \mathsf{sub1}\ (\mathsf{if}\ 0\ (\mathsf{add1}\ 7)\ 3), \emptyset, \underline{\mathsf{stop}}\rangle \\
\longmapsto_{cek} & \langle (\mathsf{if}\ 0\ (\mathsf{add1}\ 7)\ 3), \emptyset, \langle \underline{\mathsf{sub1}}\ \underline{\mathsf{stop}}\rangle\rangle \\
\longmapsto_{cek} & \langle 0, \emptyset, \langle \underline{\mathsf{if}}\ (\mathsf{add1}\ 7), 3, \emptyset, \langle \underline{\mathsf{sub1}}\ \underline{\mathsf{stop}}\rangle\rangle\rangle \\
\longmapsto_{cek} & \langle \langle \underline{\mathsf{if}}\ (\mathsf{add1}\ 7), 3, \emptyset, \langle \underline{\mathsf{sub1}}\ \underline{\mathsf{stop}}\rangle\rangle, 0\rangle \\
\longmapsto_{cek} & \langle (\mathsf{add1}\ 7), \emptyset, \langle \underline{\mathsf{sub1}}\ \underline{\mathsf{stop}}\rangle\rangle \\
\longmapsto_{cek} & \langle 7, \emptyset, \langle \underline{\mathsf{add1}}\ \langle \underline{\mathsf{sub1}}\ \underline{\mathsf{stop}}\rangle\rangle\rangle \\
\longmapsto_{cek} & \langle \langle \underline{\mathsf{add1}}\ \langle \underline{\mathsf{sub1}}\ \underline{\mathsf{stop}}\rangle\rangle, 7\rangle \\
\longmapsto_{cek} & \langle \langle \underline{\mathsf{sub1}}\ \underline{\mathsf{stop}}\rangle, 8\rangle \\
\longmapsto_{cek} & \langle \underline{\mathsf{stop}}, 7\rangle
\end{array}$$

In the first few steps, the machine adds information to the continuation in search of the first expression to evaluate. Eventually it reaches the value 0 which is returned to the continuation $\langle \underline{\mathsf{if}}\ (\mathsf{add1}\ 7), 3, \emptyset, \langle \underline{\mathsf{sub1}}\ \underline{\mathsf{stop}}\rangle\rangle$ at line 4. This continuation has all the information needed to resume the execution: based on the return value of 0, it selects the first branch of the conditional and resumes execution. The evaluation proceeds in this fashion until the machine stops with the answer 7.

As the example shows, the states of the CEK-machine have two forms:

1. $\langle M, \rho, K\rangle$ for some expression M, environment ρ, and continuation K, or
2. $\langle K, U\rangle$ for some continuation K, and some machine value U.

An *initial* state is of the form $\langle M, \emptyset, \underline{\mathsf{stop}}\rangle$. A *final* state is of the form $\langle \underline{\mathsf{stop}}, U\rangle$.

The sets ρ, U, and K are defined as follows:

$$\begin{array}{rcll}
\rho & ::= & \emptyset \mid \rho[x := U] & (\textit{Environments}) \\
U & ::= & n \mid \langle \mathbf{close}\ x, M, \rho\rangle & (\textit{Machine Values}) \\
K & ::= & \underline{\mathsf{stop}} & (\textit{Continuations}) \\
& \mid & \langle \underline{\mathsf{let}}\ x, M, \rho, K\rangle & \\
& \mid & \langle \underline{\mathsf{if}}\ M, N, \rho, K\rangle & \\
& \mid & \langle \underline{\mathsf{fun}}\ M, \rho, K\rangle & \\
& \mid & \langle \underline{\mathsf{arg}}\ U, K\rangle & \\
& \mid & \langle \underline{\mathsf{sub1}}\ K\rangle & \\
& \mid & \langle \underline{\mathsf{add1}}\ K\rangle &
\end{array}$$

The notation $\rho[x := U]$ refers to the extension of the environment ρ with the new association of x to U. We write $\rho(x)$ for the result of looking up the value of x in the environment ρ. The object $\langle \mathbf{close}\ x, M, \rho\rangle$ is the *closure* resulting from the evaluation of the procedure $(\mathsf{fun}\ x.M)$ in the environment ρ. Formally, syntactic values V are related to machine values using the following function:

$$\begin{array}{rcl}
\gamma(n, \rho) & = & n \\
\gamma(x, \rho) & = & \rho(x) \\
\gamma((\mathsf{fun}\ x.M), \rho) & = & \langle \mathbf{close}\ x, M, \rho\rangle
\end{array}$$

The set K of continuations has the same structure as the set E of evaluation contexts, and its elements have the same intuitive explanation. Every continuation is represented as a tagged record that includes the relevant machine components. Note that continuations include not only the code to execute but also the environment in which the code should be executed.

The function $\longmapsto_{cek}$ is the state transition function that specifies how the CEK-machine changes states.

$$
\begin{array}{rcll}
\langle V, \rho, K\rangle & \longmapsto_{cek} & \langle K, \gamma(V,\rho)\rangle & \\
\langle \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2, \rho, K\rangle & \longmapsto_{cek} & \langle M_1, \rho, \langle \underline{\mathsf{let}}\ x, M_2, \rho, K\rangle\rangle & \\
\langle \mathsf{if}\ M_1\ M_2\ M_3, \rho, K\rangle & \longmapsto_{cek} & \langle M_1, \rho, \langle \underline{\mathsf{if}}\ M_2, M_3, \rho, K\rangle\rangle & \\
\langle (M_1\ M_2), \rho, K\rangle & \longmapsto_{cek} & \langle M_1, \rho, \langle \underline{\mathsf{fun}}\ M_2, \rho, K\rangle\rangle & \\
\langle (\mathsf{add1}\ M), \rho, K\rangle & \longmapsto_{cek} & \langle M, \rho, \langle \underline{\mathsf{add1}}\ K\rangle\rangle & \\
\langle (\mathsf{sub1}\ M), \rho, K\rangle & \longmapsto_{cek} & \langle M, \rho, \langle \underline{\mathsf{sub1}}\ K\rangle\rangle & \\
 & & & \\
\langle\langle \underline{\mathsf{let}}\ x, M, \rho, K\rangle, U\rangle & \longmapsto_{cek} & \langle M, \rho[x := U], K\rangle & \\
\langle\langle \underline{\mathsf{if}}\ M_1, M_2, \rho, K\rangle, 0\rangle & \longmapsto_{cek} & \langle M_1, \rho, K\rangle & \\
\langle\langle \underline{\mathsf{if}}\ M_1, M_2, \rho, K\rangle, U\rangle & \longmapsto_{cek} & \langle M_2, \rho, K\rangle & \text{if } U \neq 0 \\
\langle\langle \underline{\mathsf{fun}}\ M, \rho, K\rangle, U\rangle & \longmapsto_{cek} & \langle M, \rho, \langle \underline{\mathsf{arg}}\ U, K\rangle\rangle & \\
\langle\langle \underline{\mathsf{arg}}\ \langle \mathbf{close}\ x, M', \rho'\rangle, K\rangle, U\rangle & \longmapsto_{cek} & \langle M', \rho'[x := U], K\rangle & \\
\langle\langle \underline{\mathsf{add1}}\ K\rangle, n\rangle & \longmapsto_{cek} & \langle K, m\rangle & m = n+1 \\
\langle\langle \underline{\mathsf{sub1}}\ K\rangle, n\rangle & \longmapsto_{cek} & \langle K, m\rangle & n > 0, m = n-1
\end{array}
$$

The CEK machine is slightly complicated because of its representation of continuation using tagged records. Much of this complexity can be eliminated by pre-processing the program or by uniformly representing continuations as functions [28].

4 Manipulating Continuations

Having evaluator specifications in which the continuation is explicitly represented, we can easily study how the continuation is manipulated to implement various transfers of control. To this end, we will extend TINY SCHEME with a variety of control operators and study their implementations in terms of operations on the continuation.

4.1 Error Exits

The simplest control construct is an error exit: it simply aborts the execution of the program. Formally, we extend the syntax of TINY SCHEME with one additional clause:

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

The evaluation of the expression ($\mathsf{abort}\ M$) aborts the rest of the program (*i.e.*, the continuation) and returns the value of M as the final answer of the program. For example, the following program:

add1 (abort (sub1 3))

evaluates to 2.

The semantics can be easily specified in any evaluator that has an explicit representation of the continuation. In the term rewriting semantics, it suffices to add the following clause to the definition of the one-step evaluation function:

$$E[(\mathsf{abort}\ M)] \quad \longmapsto \quad M$$

In the CEK machine, it is equally easy to specify the semantics using an additional clause in which we ignore the current continuation and use the initial continuation instead:

$$\langle(\mathsf{abort}\ M), \rho, K\rangle \quad \longmapsto_{cek} \quad \langle M, \rho, \underline{\mathsf{stop}}\rangle$$

We invite the reader to attempt to express the semantics of abort in the natural semantics of Section 3.2.

4.2 Labels and Jumps

The fundamental low-level control construct is the jump. In structured programming, the target of a jump is the closest enclosing loop header. In general, arbitrary points in the program could be marked with labels, and jumps could have label arguments specifying their targets.

To investigate the formal connection between jumps and continuations, we extend Tiny Scheme with two constructs:

$$M ::= \ldots \mid (\ell :\ M) \mid (\mathsf{jump}\ \ell\ M)$$

We assume that labels and variables are drawn from different namespaces. Informally the evaluation of $(\ell :\ M)$ defines a label ℓ that can later be jumped to, and continues with the evaluation of M. The evaluation of $(\mathsf{jump}\ \ell\ N)$ jumps backs to the point where label ℓ was encountered; the evaluation continues from that point with the value of N. For example, the program:

$$\mathsf{add1}\ (\ell :\ (\mathsf{sub1}\ (\mathsf{jump}\ \ell\ 3)))$$

evaluates to 4.

Formally the evaluation of $(\ell :\ M)$ binds ℓ to the current continuation, and the evaluation of $(\mathsf{jump}\ \ell\ N)$ ignores the current continuation and passes N to the continuation bound to ℓ instead. In terms of the CEK state transition function, the semantics can expressed using the following two clauses:

$$\begin{array}{rcl}\langle(\ell :\ M), \rho, K\rangle & \longmapsto_{cek} & \langle M, \rho[\ell := K], K\rangle \\ \langle(\mathsf{jump}\ \ell\ N), \rho, K\rangle & \longmapsto_{cek} & \langle N, \rho, \rho(\ell)\rangle\end{array}$$

Note that since the value of the label is a continuation, the set of machine values U now must include continuations:

$$U ::= n \mid \langle \mathbf{close}\ x, M, \rho\rangle \mid K \qquad (\textit{Machine Values})$$

We invite the reader to extend the term rewriting semantics and the natural semantics with rules for evaluating labels and jumps. Another interesting exercise is to specialize the semantics to the case of structured exits from loops (break/continue).

4.3 Exceptions

Even more sophisticated control constructs like exceptions correspond to simple manipulations of the continuation. We extend TINY SCHEME with a simple exception mechanism:

$$M \quad ::= \quad \ldots \mid \mathsf{throw}\ p\ V \mid \mathsf{try}\ M\ \mathsf{catch}\ p\ V$$

The informal semantics of the new expressions is as follows. In the expression ($\mathsf{throw}\ p\ V$), p is the name of a globally bound exception variable, V is a parameter to the exception. In the expression ($\mathsf{try}\ M\ \mathsf{catch}\ p\ V$), the expression V (which must be a function) is installed as an exception handler for p during the evaluation of M. If the evaluation of M throws an exception p, the exception is caught and the handler is invoked with the exception parameter as an argument. If the evaluation of M throws another exception p_1 or if the evaluation of M terminates normally, the handler is ignored.

The semantics of throw is similar to the one for abort or jump. All these constructs ignore the current continuation. The difference being that abort ignores the entire continuation, jump ignores the continuation up to its label argument, and throw ignores the continuation up to the dynamically closest handler. To express this semantics, we have two sets of evaluation contexts: general evaluation contexts E denote the continuation as before. Limited evaluation contexts E_a denote the part of the continuation that does not include a handler and hence can be safely aborted [72].

$$\begin{array}{rcl} E_a & ::= & \textit{old evaluation contexts} \\ E & ::= & E_a \mid \mathsf{try}\ E\ \mathsf{catch}\ p\ V \end{array}$$

The reduction rules are now straightforward [72]:

$$\begin{array}{rcl} E[E_a[(\mathsf{throw}\ p\ V)]] & \longmapsto & E[\mathsf{throw}\ p\ V] \\ E[\mathsf{try}\ (\mathsf{throw}\ p\ V_1)\ \mathsf{catch}\ p\ V_2] & \longmapsto & E[V_2\ V_1] \\ E[\mathsf{try}\ V_1\ \mathsf{catch}\ p\ V_2] & \longmapsto & E[V_1] \\ E[\mathsf{try}\ (\mathsf{throw}\ p_1\ V_1)\ \mathsf{catch}\ p_2\ V_2] & \longmapsto & E[\mathsf{throw}\ p_1\ V_1] \end{array}$$

The first rule aborts the part of the continuation up to the closest handler. The next three rules evaluate a try-catch expression depending on the result of evaluation of the expression in the try-block. If that expression raises the exception p, it is caught. Otherwise if the expression evaluates to a value V_1 or throws another expression p_1, the handler is ignored.

4.4 setjmp/longjmp

Another powerful control construct comes from the C library. The library routines `setjmp` and `longjmp` provide a facility similar to exception handling. Intuitively `setjmp(k)` stores the continuation in the variable `k` (which must be of the type `jump_buf` defined in the library) and returns 0 as its result. A later call to `longjmp(k,v)` ignores the current continuation and jumps back to the continuation stored in `k`; the jump transfers control back to the original `setjmp(k)` expression which now returns `v` as its result. For example, the following program prints `f(5) = 42`:

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf k;

int f (int i) {
  if (i == 0) longjmp(k, 42);
  else return i*f(i-1);
}

void main () {
  int result = setjmp(k);
  switch (result) {
  case 0: f(5); break; // setjmp returns 0 first time
  default: printf("f(5) = %d\n", result); break; // we get here by longjmp
  }
}
```

It should be clear that the semantics of `setjmp/longjmp` is similar to the semantics of jumps/labels and exceptions presented earlier, and could in principle be formalized in the same manner. However, the similarity is only superficial: there is a rather subtle point about the `setjmp/longjmp` constructs. The call `setjmp(k)` does not actually copy the continuation in `k`. In particular, although the continuation includes the entire control stack, only the stack pointer is copied. Hence, a `longjmp(k,v)` only makes sense at a point where the control stack is bigger compared to the point where the `setjmp` occurred, otherwise we have a dangling stack pointer. The invariant needed for the correct operation of these constructs is not enforced by the C language or library and unpredictable results can occur from the undisciplined use of `setjmp/longjmp`. For example, the execution of the following program causes a segmentation fault on most implementations.

```
#include <setjmp.h>

jmp_buf k;

int f (int i) {
  if (i == 0) return setjmp(k);
  else return i*f(i-1);
}

void main () {
  int result = f(5);
  longjmp(k,42);
}
```

4.5 Forward Jumps

The limitation of `setjmp/longjmp` raises the question of more general *forward* jumps. To understand the difference between such jumps and conventional jumps, we recall our intuitive correspondence between continuations and control stacks.

One can view the evaluation of a labeled expression as setting a mark on the control stack that can be later jumped to. Most languages restrict labels and jumps to ensure that jumps can only occur while the mark is still on the stack. Once the activation record containing the mark is popped, it is either impossible to jump or the result of the jump is undefined. In summary, a conventional jump eliminates part of an existing control stack. Similar restrictions apply to exceptions and handlers.

Consider however an alternative implementation of a labeled expression. Instead of simply setting a mark on the control stack, we could copy the entire stack to an alternate location. A jump would then consist of replacing the current control stack by the stored stack associated with the label. This implementation would work whether the current stack includes the mark or not. Despite its apparent cost, this implementation can be considerably optimized to avoid unbounded or useless copying of the stack [38]. (See [8] for a survey of implementation strategies.)

The reader may wonder at this point about the usefulness of forward jumps. The next section will provide some examples. Meanwhile, we invite the reader to modify the extension of Tiny Scheme with labels and jumps in Section 4.2 to accomodate forward jumps. Note that the rule for $\ell :\ M$ already copies the entire continuation:

$$\langle (\ell :\ M), \rho, K \rangle \longmapsto_{cek} \langle M, \rho[\ell := K], K \rangle$$

One way to extend the language with forward jumps is to treat labels as first-class values that may be passed as arguments and returned as results [53].

5 A Unified Framework of Control

The previous section illustrates that many control operators are merely simple operations on the continuation. Instead of providing a large library of control abstractions for Tiny Scheme, it is tempting to extend Tiny Scheme once and for all with an ultimate control construct, upon which users can build their own abstractions.

We therefore extend Tiny Scheme with the control construct *call-with-current-continuation* (abbreviated *call/cc*) [7, 9, 42, 1]. Intuitively, the expression:

$$call/cc\ (\mathsf{fun}\ k.e)$$

binds k to a functional representation of the current continuation. This continuation can be freely used in the expression e as any other function: in particular it may be returned to an outer scope as part of the result of e. A call to k from anywhere in the program (inside or outside e) would cause control to jump back to the point where *call/cc* was originally executed.

The similarity of the above description with the semantics of jumps and labels, exceptions, `setjmp/longjmp` should suggest that *call/cc* can easily implement such constructs.

In addition, *call/cc* can express some additional sophisticated control abstractions including threads [5, 60, 37, 71], engines [36, 15], coroutines [33], backtracking [29], Prolog-style control [62, 16, 34], and recursion [26].

5.1 Call-with-current-continuation

By far the most common continuation-based control operator is *call/cc*. There are, however, many other continuation-based operators, that differ from *call/cc* in subtle aspects [25, 35, 54, 67, 68]. Some operators are even more powerful than *call/cc* and allow continuations to be delimited and composed [12, 19, 21, 39, 40, 61]. We concentrate on the simpler and more widely used *call/cc*.

The syntax of TINY SCHEME is augmented with a additional syntactic value:

$$V ::= \ldots \mid \mathit{call/cc}$$

Based on our discussion, the following program:

$$\mathsf{add1}\ (\mathit{call/cc}\ (\mathsf{fun}\ k.(\mathsf{sub1}\ (k\ 3))))$$

should evaluate to 4 as follows. First the continuation which consists of the addition is bound to k. Then evaluation proceeds until it reaches the application of k to 3. At this point, the current continuation which includes the subtraction is ignored. Instead the continuation bound to k is used. This continuation consists of only the addition and hence the final result is 4.

The semantics of *call/cc* is easy to formalize using either the CEK machine [22] or term rewriting [18, 20, 23, 24]. The CEK machine needs to be augmented with a clause for capturing the continuation that should look familiar by now:

$$\langle \mathit{call/cc}\ (\mathsf{fun}\ k.M), \rho, K\rangle \longmapsto_{cek} \langle M, \rho[k := K], K\rangle$$

As for labels, the set of machine values must include continuations. A call to the continuation is handled by the following clause:

$$\langle\langle \underline{\mathsf{arg}}\ K', K\rangle, U\rangle \longmapsto_{cek} \langle K', U\rangle$$

To illustrate the clauses in action, we trace the evaluation of our simple example:

$$\begin{array}{ll}
 & \langle \mathsf{add1}\ (\mathit{call/cc}\ (\mathsf{fun}\ k.(\mathsf{sub1}\ (k\ 3)))), \emptyset, \underline{\mathsf{stop}}\rangle \\
\longmapsto_{cek} & \langle \mathit{call/cc}\ (\mathsf{fun}\ k.(\mathsf{sub1}\ (k\ 3))), \emptyset, \langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle\rangle \\
\longmapsto_{cek} & \langle \mathsf{sub1}\ (k\ 3), \emptyset[k := (\langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle)], \langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle\rangle \\
\longmapsto_{cek} & \langle (k\ 3), \emptyset[k := (\langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle)], \langle \underline{\mathsf{sub1}}\ \langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle\rangle\rangle \\
\longmapsto_{cek} & \langle k, \emptyset[k := (\langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle)], \langle \underline{\mathsf{fun}}\ 3, \emptyset[k := (\langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle)], \langle \underline{\mathsf{sub1}}\ \langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle\rangle\rangle\rangle \\
\longmapsto_{cek} & \langle\langle \underline{\mathsf{fun}}\ 3, \emptyset[k := (\langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle)], \langle \underline{\mathsf{sub1}}\ \langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle\rangle\rangle, (\langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle)\rangle \\
\longmapsto_{cek} & \langle 3, \emptyset[k := (\langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle)], \langle \underline{\mathsf{arg}}\ (\langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle), \langle \underline{\mathsf{sub1}}\ \langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle\rangle\rangle\rangle \\
\longmapsto_{cek} & \langle\langle \underline{\mathsf{arg}}\ (\langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle), \langle \underline{\mathsf{sub1}}\ \langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle\rangle\rangle, 3\rangle \\
\longmapsto_{cek} & \langle (\langle \underline{\mathsf{add1}}\ \underline{\mathsf{stop}}\rangle), 3\rangle \\
\longmapsto_{cek} & \langle \underline{\mathsf{stop}}, 4\rangle
\end{array}$$

The semantics can also be expressed in the term rewriting approach but requires a small extension in order to be able to find a syntactic representation for continuation functions. Unlike normal functions, continuation functions cause a jump. It turns out that we can express this jumping ability of continuations using abort. Thus, for the purposes of semantic description, we assume the existence of abort. A single rule is now sufficient to explain the semantics of *call/cc*:

$$E[\mathit{call/cc}\ (\mathsf{fun}\ k.M)] \longmapsto E[M[k := (\mathsf{fun}\ x.(\mathsf{abort}\ E[x]))]]$$

In the rule, the continuation at the point when *call/cc* is executed is represented by E. So in principle, we should bind k to E. However, we must also ensure that when k is called, the continuation at the point of the call is ignored and replaced by E. To achieve this effect, we surround the use of E with an abort. To illustrate these points in more detail, let's trace the evaluation of our simple example:

$$\begin{array}{ll}
& \mathsf{add1}\ (\mathit{call/cc}\ (\mathsf{fun}\ k.(\mathsf{sub1}\ (k\ 3)))) \\
\longmapsto & \mathsf{add1}\ (\mathsf{sub1}\ ((\mathsf{fun}\ x.(\mathsf{abort}\ (\mathsf{add1}\ x)))\ 3)) \\
\longmapsto & \mathsf{add1}\ (\mathsf{sub1}\ (\mathsf{abort}\ (\mathsf{add1}\ 3))) \\
\longmapsto & \mathsf{add1}\ (\mathsf{sub1}\ (\mathsf{abort}\ 4)) \\
\longmapsto & 4
\end{array}$$

We invite the reader to explore forward jumps using *call/cc*.

5.2 Pragmatics

As promised, we show that *call/cc* can implement common control abstractions that are usually provided as built-in extensions or in separate libraries in other languages.

Labels and Jumps The correspondence with labels and jumps is straightforward. Defining a label is like capturing the continuation and binding it to a variable. Jumping to a label is like calling a continuation. Hence, one can express labels and jumps as follows:

$$\begin{array}{rcl}
(\ell :\ M) & \equiv & (\mathit{call/cc}\ (\mathsf{fun}\ \ell.M) \\
(\mathsf{jump}\ \ell\ M) & \equiv & (\ell\ M)
\end{array}$$

Exceptions The implementation of an exception mechanism using *call/cc* cannot be directly expressed in TINY SCHEME as it lacks the assignments necessary to implement the dynamic scope of handlers. We therefore use full Scheme in the following discussion.

Assuming for simplicity that we only have one exception name, then a simple implementation of exceptions in Scheme is the following. First, we reserve some global location which will contain the current handler for the exception. Uses of throw simply call the current handler from the global location. Uses of try-catch update the global location with a new handler taking care to save the old handler first. The old handler must be restored before the try-catch expression returns as well as before handling any exceptions. To enable uses of throw to abort the continuation up to the handler, the handler is always surrounded with a continuation. Putting everything together, we get:

```
(define global_handler_location ;; initially no handler is installed
  (lambda (v)
    (error 'global_handler_location "Uncaught exception")))

(define throw ;; just call the current handler
  (lambda (v)
    (global_handler_location v)))

(define try_catch
  (lambda (body handler)
    ;; if the handler is called, it returns here
    (call/cc (lambda (k) ;; first mark the current point
      (let ((old_handler global_handler_location)) ;; save old handler
        (set! global_handler_location
              ;; Update the location with a new handler.
              (lambda (v)
                ;; When the new handler is called, it first restores the
                ;; old handler, then handles the exception by JUMPING to
                ;; the point where the try_catch was executed aborting
                ;; any intermediate computations.
                (set! global_handler_location old_handler)
                (k (handler v))))
        (let ((result (body))) ;; evaluate the body
          (set! global_handler_location old_handler) ;; restore handler
          ;; normal return
          result))))))
```

As an illustration, consider the following simple example:

```
(define div
  (lambda (x y)
    (if (zero? y)
        (throw 0)
        (/ x y))))

(define atan_estimate
  (lambda (side1 side2)
    (try_catch
     (lambda () (let ((ratio (div side1 side2)))
                  (if (< ratio 1)
                      "0-45 degrees"
                      (if (> ratio 1)
                          "45-90 degrees"
                          "45 degrees"))))
     (lambda (v) "90 degrees"))))
```

```
(define main
  (lambda ()
    (list
     (atan_estimate 1 1)
     (atan_estimate 1 0)
     (atan_estimate 1 2)
     (atan_estimate 2 1))))
```

The function `div` performs ordinary division but throws an exception if the denominator is 0. The function `atan_estimate` takes the lengths of two sides of a right triangle and estimates the angle who tangent is given by `side1/side2`. If the ratio is a number then the angle is estimated according to the familiar formula. If the division throws an exception, then the ratio is infinity, and the angle must be 90 degrees. Indeed, running `main()` prints:

```
("45 degrees" "90 degrees" "0-45 degrees" "45-90 degrees")
```

6 Programming with Continuations

Not only can *call/cc* define general-purpose abstractions like coroutines and threads, but it can also be used to define customized abstractions that are unlikely to make it in a general-purpose programming language. We illustrate this idea by solving a problem adapted from an interpreter by Dan Friedman [11].

The problem consists of a pre-order traversal of a *spirited binary tree* which contains three special kinds of nodes: milestones, devils, and angels. The general rule is that a devil sends you back to the latest milestone; an angel sends you back to the latest devil. For concretness, here is the core of a Scheme implementation using call/cc:

```
(define visitTree
  (lambda (tree)
    (if (leaf? tree)
        (void)
        (call/cc (lambda (k)
          (visitNode (getNode tree) k)
          (visitTree (getLeft tree))
          (visitTree (getRight tree)))))))

(define visitNode
  (lambda (node k)
    (printf "~s~n" node)
    (case node
      [(milestone) (pushMilestone k)]
      [(devil)     (pushDevil k) (popMilestone)]
      [(angel)     (popDevil)]
      [else        (void)])))
```

The code uses some simple auxiliary functions to manipulate the tree data structure, and two stacks: one for milestone nodes and one for devil nodes.

The Scheme implementation reveals an extremely important point: the structure of `visitTree` is the same as the standard recursive pre-order traversal of a binary tree. Thus one can start from a well-understood traversal and just add code to manipulate the continuation when visiting a node.

In contrast, when writing in a language without *call/cc* or any similar operator, programmers have two choices. Those that are aware of the CPS transformation can systematically rewrite the Scheme solution to eliminate *call/cc*.

Those that are not familiar with the CPS transformation are unlikely to stumble upon it and are more likely to develop an *ad hoc* solution to this problem. Any solution that starts with a recursive pre-order traversal of the tree is "wrong." Indeed consider the situation in which we encounter a milestone after a number A of recursive calls, and then after B more recursive calls, we encounter a devil. We should now somehow immediately return from the last B recursive calls and resume the computation after the first A recursive calls. Unfortunately most programming languages do not offer this kind of flexibility in the manipulation of the run-time stack. Thus when using a language like C, one usually manages the stack explicitly by writing an iterative version of the pre-order traversal that uses a loop. Using this idea, one can now write a correct (but a bit convoluted) solution to the problem. We invite the reader to try.

In general, continuations provide elegant solutions to a large number of problems [29].

7 Conclusion: The Future of Continuations

We have offered an understanding of continuations based on programming practice. This understanding should prove useful in the most recent trend based on applets and mobile code. The connection between continuations and control stacks generalizes in two ways. In a concurrent setting, continuations describe threads and are useful for programming multi-threaded graphical user interfaces [31] or simulations. In a distributed setting, continuations describe processor states. To move a computation, one simply packages the continuation and sends it to a remote processor [70].

Continuations truly provide a universal means of understanding *all* transfers of control, starting from goto all the way to mobile threads.

References

[1] Abelson, H., Dybvig, R. K., Haynes, C. T., Rozas, G. J., IV, N. I. A., Friedman, D. P., Kohlbecker, E., Steele Jr., G. L., Bartley, D. H., Halstead, R., Oxley, D., Sussman, G. J., Brooks, G., Hanson, C., Pitman, K. M., and Wand, M. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11*, 1 (Aug. 1998), 7–105.

[2] Aho, A., Sethi, R., and Ullman, J. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1985.

[3] Appel, A. *Compiling with Continuations.* Cambridge University Press, 1992.

[4] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed., vol. 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers B.V., Amsterdam, 1984.

[5] Biagioni, E., Cline, K., Lee, P., Okasaki, C., and Stone, C. Safe-for-space threads in standard ml. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations* (Jan. 1997), O. Danvy, Ed. BRICS Notes Series NS-96-13.

[6] Boehm, H.-J. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems 7*, 4 (Oct. 1985), 637–655.

[7] Clinger, W., Friedman, D., and Wand, M. A scheme for a higher-level semantic algebra. In *Algebraic Methods in Semantics*, J. C. Reynolds and M. Nivat, Eds. Cambridge University Press, 1985, pp. 237–250.

[8] Clinger, W., Hartheimer, A., and Ost, E. Implementation strategies for continuations. In *ACM Conference on Lisp and Functional Programming* (1988), pp. 124–131.

[9] Clinger, W., and Rees, J. Revised^4 report on the algorithmic language Scheme. *Lisp Pointers 4*, 3 (1991), 1–55.

[10] Clinger, W. D. Proper tail recursion and space efficiency. *ACM SIGPLAN Notices 33*, 5 (May 1998), 174–185.

[11] D. P. Friedman, D. Applications of continuations. Tech. Rep. 237, Indiana Univeristy Computer Science Department, 1988.

[12] Danvy, O., and Filinski, A. Abstracting control. In *ACM Conference on Lisp and Functional Programming* (1990), pp. 151–160.

[13] Danvy, O., and Filinski, A. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science 2*, 4 (1992), 361–391.

[14] Draves, R. P. *Control Transfer in Operating System Kernels.* PhD thesis, Carnegie-Mellon University, Department of Computer Science, May 1994. Available as Tech. Rep. CMU-CS-94-142.

[15] Dybvig, K., and Hieb, R. Engines from continuations. *Journal of Computer Languages* (Pergamon Press) *14*, 2 (1989), 109–124.

[16] Felleisen, M. Transliterating prolog into scheme. Tech. Rep. 182, Indiana University Computer Science Department, 1985.

[17] Felleisen, M. *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.

[18] Felleisen, M. λ-v-CS: An extended λ-calculus for Scheme. In *ACM Conference on Lisp and Functional Programming* (1988), pp. 72–84.

[19] Felleisen, M. The theory and practice of first-class prompts. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1988), pp. 180–190.

[20] Felleisen, M., and Friedman, D. A reduction semantics for imperative higher-order languages. In *Conference on Parallel Architectures and Languages* (1987), Lecture Notes in Computer Science, 259, pp. 206–223.

[21] Felleisen, M., Friedman, D., Duba, B. F., and Merrill, J. Beyond continuations. Tech. Rep. 216, Indiana University Computer Science Department, 1987.

[22] Felleisen, M., and Friedman, D. P. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts-III* (1986), M. Wirsing, Ed., North-Holland, pp. 193–217.

[23] Felleisen, M., Friedman, D. P., Kohlbecker, E., and Duba, B. A syntactic theory of sequential control. *Theoret. Comput. Sci. 52*, 3 (1987), 205–237. Preliminary version: Reasoning with Continuations, in Proceedings of the 1st IEEE Symposium on Logic in Computer Science, 1986.

[24] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci. 102* (1992), 235–271. Tech. Rep. 89-100, Rice University.

[25] Felleisen, M., Wand, M., Friedman, D., and Duba, B. Abstract continuations: A mathematical semantics for handling full functional jumps. In *ACM Conference on Lisp and Functional Programming* (1988), pp. 52–62.

[26] Filinski, A. Recursion from iteration. Tech. Rep. CS-92-1426, Stanford University, 1992. ACM Sigplan Workshop on Continuations, 3-11.

[27] Fischer, M. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions About Programs* (1972), Sigplan Notices, 7, 1, pp. 104–109. Revised version in *Lisp and Symbolic Computation*, **6**, 3/4, (1993) 259-287.

[28] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In the *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1993), pp. 237–247.

[29] Friedman, D. P., Haynes, C., and Kohlbecker, E. Programming with continuations. In *Program Transformations and Programming Environments*, P. Pepper, Ed. Springer-Verlag, 1985, pp. 263–274.

[30] Friedman, D. P., Wand, M., and Haynes, C. *Essentials of Programming Languages.* The MIT Press, 1992.

[31] Fuchs, M. Escaping the event loop: An alternative control structure for multi-threaded guis. In *Proceedings of EHCI* (London, 1995), Chapman-Hall.

[32] Griffin, T. A formulae-as-types notion of control. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1990), pp. 47–58.

[33] Haynes, C., Friedman, D. P., and Wand, M. Obtaining coroutines from continuations. *Journal of Computer Languages* (Pergamon Press) *11*, 3/4 (1986), 143–153.

[34] Haynes, C. T. Logic continuations. *Journal of Logic Programming 4* (1987), 157–176. Preliminary version in Proceedings of the Third International Conference on Logic Programming, 1985, Lecture Notes in Computer Science, 225.

[35] Haynes, C. T., and Friedman, D. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems 9*, 4 (1987), 582–598. Preliminary version: Constraining Control. In Conference Record of the 12th ACM Symposium on Principles of Programming Languages, 1985.

[36] Haynes, C. T., and Friedman, D. P. Engines build process abstractions. In *ACM Conference on Lisp and Functional Programming* (1984), pp. 18–24.

[37] Hieb, R., and Dybvig, K. Continuations and concurrency. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (1990), pp. 128–136.

[38] Hieb, R., Dybvig, K., and Bruggeman, C. Representing control in the presence of first-class continuations. In the *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1990), pp. 66–77.

[39] Johnson, G. F. GL—a language and environment for interactively experimenting with denotational definitions. In *Proceedings of the SIGPLAN Symposium on Interpreters and Interpretive Techniques* (1987), Sigplan Notices, 22, 7, pp. 165–176.

[40] Johnson, G. F., and Duggan, D. Stores and partial continuations as first-class objects in a language and its environment. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1988), pp. 158–168.

[41] Kahn, G. Natural semantics. In *Proc. STACS* (1987), vol. 247 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 22–39.

[42] Kelsey, R., Clinger, W., and Rees, J. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (Sept. 1998), 26–76. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[43] Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. Orbit: An optimizing compiler for Scheme. In *ACM SIGPLAN Symposium on Compiler Construction* (1986), Sigplan Notices, 21, 7, pp. 219–233.

[44] Landin, P. The mechanical evaluation of expressions. *Comput. J. 6*, 4 (1964), 308–320.

[45] Landin, P. J. An abstract machine for designers of computing languages. In *Proceedings IFIP Congress* (1965), pp. 438–439.

[46] Mason, I., and Talcott, C. L. Inferring the equivalence of functional programs that mutate data. *Theoret. Comput. Sci. 105*, 2 (1992), 167–215. Preliminary version in Proceedings of the 4th IEEE Symposium on Logic in Computer Science 1989.

[47] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[48] Milner, R., Tofte, M., Harper, R., and MacQueen, D. B. *The Standard ML Programming Language (Revised).* MIT Press, 1997.

[49] Murthy, C. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell, 1990.

[50] Murthy, C. An evaluation semantics for classical proofs. In the *IEEE Symposium on Logic in Computer Science* (1991).

[51] Parigot, M. Classical proofs as programs. In *Computational Logic and Proof Theory: Proceedings of the third Godel Colloquium* (1993), Lecture Notes in Computer Science (713), pp. 263–276.

[52] Plotkin, G. D. Call-by-name, call-by-value, and the λ-calculus. *Theoret. Comput. Sci. 1* (1975), 125–159.

[53] Reynolds, J. C. Gedanken—a simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM 13*, 5 (1970), 308–319.

[54] Reynolds, J. C. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference* (1972), pp. 717–740.

[55] Reynolds, J. C. The discoveries of continuations. *Lisp Symbol. Comput. 6*, 3/4 (1993), 233–247.

[56] Rozas, G. Liar, an Algol-like compiler for Scheme. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 1984.

[57] Sabry, A., and Felleisen, M. Reasoning about programs in continuation-passing style. *Lisp Symbol. Comput. 6*, 3/4 (1993), 289–360. Also in the *ACM Conference on Lisp and Functional Programming* (1992) and Tech. Rep. 92-180, Rice University.

[58] Sabry, A., and Field, J. Reasoning about explicit and implicit representations of state. Tech. Rep. YALEU/DCS/RR-968, Yale University, 1993. *ACM SIGPLAN Workshop on State in Programming Languages*, pages 17-30.

[59] Sabry, A., and Wadler, P. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems 19*, 6 (Nov. 1997), 916–941.

[60] Shivers, O. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations* (Jan. 1997), O. Danvy, Ed. BRICS Notes Series NS-96-13.

[61] Sitaram, D., and Felleisen, M. Control delimiters and their hierarchies. *Lisp Symbol. Comput. 3*, 1 (1990), 67–100.

[62] Srivastra, A., Oxley, D., and Srivastra, D. An(other) integration of logic and functional programming. In *Proceedings IEEE Symposium on Logic Programming* (1985), pp. 254–260.

[63] Steele, G. L. Rabbit: A compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, 1978.

[64] Steele, Jr., G. L. Debunking the "expensive procedure call" myth. AI Memo 443, MIT Artificial Intelligence Laboratory, Oct. 1977. Alternate Titles: "Procedure call implementation considered Harmful" or "LAMBDA: the ultimate GOTO".

[65] Stoy, J. *Denotational Semantics: The Scott-Strachey Approach to Programming Languages.* The MIT Press, Cambridge, Mass., 1981.

[66] Strachey, C., and Wadsworth, C. Continuations: A mathematical semantics for handling full jumps. Technical monograph prg-11, Oxford University Computing Laboratory, Programming Research Group, 1974.

[67] Talcott, C. L. *The Essence of Rum—A Theory of the Intensional and Extensional Aspects of Lisp-Type Computation.* PhD thesis, Stanford University, 1985.

[68] Talcott, C. L. Programming and proving with function and control abstractions. Lecture Notes for the Western Institute of Computer Science, Standford, manuscript, 1987.

[69] Talcott, C. L. A theory for program and data specification. *Theoret. Comput. Sci. 104* (1992), 129–159. Preliminary version in Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems, (Lecture Notes in Computer Science, 429, 1990).

[70] Tarau, P., and Dahl, V. Mobile threads through first order continuations. In *Proceedings of APPAI-GULP-PRODE* (Coruna, Spain, July 1998).

[71] Wand, M. Continuation-based multiprocessing. In *ACM Conference on Lisp and Functional Programming* (1980), pp. 19–28.

[72] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. *Inf. Comput. 115*, 1 (Nov. 1994), 38–94.