# Chapter 1

# Continuations

In the simply typed $\lambda$-calculus, a complete reduction of $(\lambda x\!:\!A.\,e)\ v$ to another value $v'$ consists of a sequence of $\beta$-reductions. From the perspective of imperative languages, the complete reduction consists of two *local* transfers of control: a function call and a return. We may think of a $\beta$-reduction $(\lambda x\!:\!A.\,e)\ v \mapsto [v/x]e$ as initiating a call to $\lambda x\!:\!A.\,e$ with an argument $v$, and $[v/x]e \mapsto^* v'$ as returning from the call with the result $v'$.

This chapter investigates language constructs for achieving *non-local* transfers of control in the simply typed $\lambda$-calculus. By non-local transfers of control, we mean those reductions that cannot be justified by $\beta$-reductions. The key concept behind non-local transfers of control is *continuations*, which may be thought of as generalizations of evaluation contexts in Chapter **??**. The basic idea is that evaluation contexts are turned into first-class objects which can be passed as arguments to functions or return values of functions. More importantly, an evaluation context elevated to a first-class object may replace the current evaluation context, thereby achieving a non-local transfer of control.

Continuations in the simply typed $\lambda$-calculus are often compared to the goto construct of imperative languages. Like the goto construct, continuations are a powerful control construct whose applications range from a simple optimization of list multiplication (to be discussed in Section 1.1) to an elegant implementation of the machinery for concurrent computations. On the other side of the coin, continuations are often detrimental to code readability, and should be used with great care for the same reason that the goto construct is avoided in favor of loop constructs in imperative languages.

## 1.1 A motivating example

A prime example for motivating the development of continuations is a recursive function for list multiplication, *i.e.*, for multiplying all elements in a given list. Let us begin with an SML function implementing list multiplication:

```
fun multiply l =
  let
    fun mult nil = 1
      | mult (n :: l') = n * mult l'
  in
    mult l
  end
```

We wish to optimize `multiply` by exploiting the property that in the presence of a zero in `l`, the return value of `multiply` is also a zero regardless of other elements in `l`. Thus, once we encounter an occurrence of a zero in `l`, we do not have to multiply elements in the remaining list:

```
fun multiply' l =
  let
    fun mult nil = 1
      | mult (0 :: l') = 0
      | mult (n :: l') = n * mult l'
  in
    mult l
  end
```

`multiply'` is definitely an improvement over `multiply`, although if `l` contains no zero, it runs slower than `multiply` because of the cost of comparing each element in `l` with `0`. `multiply'`, however, is not a full optimization of `multiply` exploiting the property of multiplication: due to the recursive nature of `mult`, it needs to return a zero as many times as the number of elements before the first zero in `l`. Thus an ideal solution would be to exit `mult` altogether after encountering a zero in `l`, *even without returning a zero to previous calls to* `mult`. What makes this possible is two constructs, `callcc` and `throw`, for continuations:[1]

```
fun multiply'' l =
  callcc (fn ret =>
  let
    fun mult nil = 1
      | mult (0 :: l') = throw ret 0
      | mult (n :: l') = n * mult l'
  in
    mult l
  end)
```

Informally `callcc (fn ret => ...` declares a label `ret`, and `throw ret 0` causes a non-local transfer of control to the label `ret` where the evaluation resumes with a value `0`. Hence there occurs no return from `mult` once `throw ret 0` is reached.

Below we give a formal definition of the two constructs `callcc` and `throw`.

## 1.2 Evaluation contexts as continuations

A continuation is a general concept for describing an "incomplete" computation which yields a "complete" computation only when another computation is *prepended* (or *prefixed*).[2] That is, by joining a computation with a continuation, we obtain a complete computation. A $\lambda$-abstraction $\lambda x\!:\!A.\,e$ may serve as a continuation, since it conceptually takes a computation producing a value $v$ and returns a computation corresponding to $[v/x]e$. Note that $\lambda x\!:\!A.\,e$ itself does not initiate a computation; it is only when an argument $v$ is supplied that it initiates a computation of $[v/x]e$. An excellent example of continuation is an evaluation context $\kappa$ which, given an expression $e$, yields a computation corresponding to $\kappa[\![e]\!]$. Like a $\lambda$-abstraction, $\kappa$ itself does not describe a complete computation. In this chapter, we restrict ourselves to evaluation contexts as a means of studying continuations.

Consider the rule $\mathit{Red}_\beta$ which decomposes a given expression into a unique evaluation context $\kappa$ and a unique subexpression $e$:

$$\frac{e \mapsto_\beta e'}{\kappa[\![e]\!] \mapsto \kappa[\![e']\!]}\ \mathit{Red}_\beta$$

Since the decomposition under the rule $\mathit{Red}_\beta$ is implicit and evaluation contexts are not expressions, there is no way to store $\kappa$ as an expression. Hence our first goal is to devise a new construct for seizing the current

[1] In SML/NJ, open the structure `SMLofNJ.Cont` to test `multiply''`.

[2] Here "prepend" and "prefix" both mean "adding to the beginning."

evaluation context.[3] For example, when a given expression is decomposed into $\kappa$ and $e$ by the rule $Red_\beta$, the new construct would return a (new form of) value storing $\kappa$. The second goal is to involve such a value in a reduction sequence, as there is no point in creating such a value without using it.

In order to utilize evaluation contexts as continuations in the simply typed $\lambda$-calculus, we introduce three new constructs: $\langle\kappa\rangle$, $\mathsf{callcc}\ x.\,e$, and $\mathsf{throw}\ e\ \mathsf{to}\ e'$.

- $\langle\kappa\rangle$ is an expression storing an evaluation context $\kappa$; we use angle brackets $\langle\rangle$ to distinguish it as an expression not to be confused with an evaluation context. The only way to generate it is by reducing $\mathsf{callcc}\ x.\,e$. As a value, $\langle\kappa\rangle$ is called a continuation.
- $\mathsf{callcc}\ x.\,e$ seizes the current evaluation context $\kappa$ and stores $\langle\kappa\rangle$ in $x$ before proceeding to reduce $e$:

$$\frac{}{\kappa[\![\mathsf{callcc}\ x.\,e]\!] \mapsto \kappa[\![[\langle\kappa\rangle/x]e]\!]}\ \mathit{Callcc}$$

  In the case that the reduction of $e$ does not use $x$ at all, $\mathsf{callcc}\ x.\,e$ produces the same result as $e$.
- $\mathsf{throw}\ e\ \mathsf{to}\ e'$ expects a value $v$ from $e$ and a continuation $\langle\kappa'\rangle$ from $e'$. Then it starts a reduction of $\kappa'[\![v]\!]$ *regardless of the current evaluation context* $\kappa$:

$$\frac{}{\kappa[\![\mathsf{throw}\ v\ \mathsf{to}\ \langle\kappa'\rangle]\!] \mapsto \kappa'[\![v]\!]}\ \mathit{Throw}$$

  In general, $\kappa$ and $\kappa'$ are unrelated with each other, which implies that the rule *Throw* allows us to achieve a non-local transfer of control. We say that $\mathsf{throw}\ v\ \mathsf{to}\ \langle\kappa'\rangle$ *throws* a value $v$ to a continuation $\kappa'$.

The abstract syntax is extended as follows:

$$\begin{array}{rrcl}
\mathsf{expression} & e & ::= & \cdots \mid \mathsf{callcc}\ x.\,e \mid \mathsf{throw}\ e\ \mathsf{to}\ e \mid \langle\kappa\rangle \\
\mathsf{value} & v & ::= & \cdots \mid \langle\kappa\rangle \\
\mathsf{evaluation\ context} & \kappa & ::= & \cdots \mid \mathsf{throw}\ \kappa\ \mathsf{to}\ e \mid \mathsf{throw}\ v\ \mathsf{to}\ \kappa
\end{array}$$

The use of evaluation contexts $\mathsf{throw}\ \kappa\ \mathsf{to}\ e$ and $\mathsf{throw}\ v\ \mathsf{to}\ \kappa$ indicates that $\mathsf{throw}\ e\ \mathsf{to}\ e'$ reduces $e$ before reducing $e'$.

**Exercise 1.1.** What is the result of evaluating each expression below?

$$\begin{array}{lllc}
(1) & \mathsf{fst}\ \mathsf{callcc}\ x.\,(\mathsf{true}, \mathsf{false}) & \mapsto^* & ? \\
(2) & \mathsf{fst}\ \mathsf{callcc}\ x.\,(\mathsf{true}, \mathsf{throw}\ (\mathsf{false}, \mathsf{false})\ \mathsf{to}\ x) & \mapsto^* & ? \\
(3) & \mathsf{snd}\ \mathsf{callcc}\ x.\,(\mathsf{throw}\ (\mathsf{true}, \mathsf{true})\ \mathsf{to}\ x, \mathsf{false}) & \mapsto^* & ?
\end{array}$$

In the case (1), $x$ is not found in $(\mathsf{true}, \mathsf{false})$, so the expression is equivalent to $\mathsf{fst}\ (\mathsf{true}, \mathsf{false})$. In the case (2), the result of evaluating $\mathsf{true}$ is eventually ignored because the reduction of $\mathsf{throw}\ (\mathsf{false}, \mathsf{false})\ \mathsf{to}\ x$ causes $(\mathsf{false}, \mathsf{false})$ to replace $\mathsf{callcc}\ x.\,(\mathsf{true}, \mathsf{throw}\ (\mathsf{false}, \mathsf{false})\ \mathsf{to}\ x)$. Thus, in general, $\mathsf{fst}\ \mathsf{callcc}\ x.\,(e, \mathsf{throw}\ (\mathsf{false}, e')\ \mathsf{to}\ x)$ evaluates to $\mathsf{false}$ regardless of $e$ and $e'$ (provided that the evaluation terminates). In the case (3), $\mathsf{false}$ is not even evaluated: before reaching $\mathsf{false}$, the reduction of $\mathsf{throw}\ (\mathsf{true}, \mathsf{true})\ \mathsf{to}\ x$ causes $(\mathsf{true}, \mathsf{true})$ to replace $\mathsf{callcc}\ x.\,(\mathsf{throw}\ (\mathsf{true}, \mathsf{true})\ \mathsf{to}\ x, \mathsf{false})$. Thus, in general, $\mathsf{snd}\ \mathsf{callcc}\ x.\,(\mathsf{throw}\ (e, \mathsf{true})\ \mathsf{to}\ x, e')$ evaluates to $\mathsf{true}$ regardless of $e$ and $e'$, where $e'$ is never evaluated:

$$\begin{array}{lllc}
(1) & \mathsf{fst}\ \mathsf{callcc}\ x.\,(\mathsf{true}, \mathsf{false}) & \mapsto^* & \mathsf{true} \\
(2) & \mathsf{fst}\ \mathsf{callcc}\ x.\,(e, \mathsf{throw}\ (\mathsf{false}, e')\ \mathsf{to}\ x) & \mapsto^* & \mathsf{false} \\
(3) & \mathsf{snd}\ \mathsf{callcc}\ x.\,(\mathsf{throw}\ (e, \mathsf{true})\ \mathsf{to}\ x, e') & \mapsto^* & \mathsf{true}
\end{array}$$

Now that we have seen the reduction rules for the new constructs, let us turn our attention to their types. Since $\langle\kappa\rangle$ is a new form of value, we need a new form of type for it. (Otherwise how do we represent

[3] I hate the word *seize* because the *z* sound in it is hard to enunciate. Besides I do not want to remind myself of *Siege* Tanks in Starcraft!

its type?) We assign a type $A$ cont to $\langle\kappa\rangle$ if the hole in $\kappa$ expects a value of type $A$. That is, if $\kappa : A \Rightarrow C$ holds (see Section **??**), $\langle\kappa\rangle$ has type $A$ cont:

$$\text{type} \qquad A \quad ::= \quad \cdots \mid A \; \mathsf{cont}$$

$$\frac{\kappa : A \Rightarrow C}{\Gamma \vdash \langle\kappa\rangle : A \; \mathsf{cont}} \; \mathsf{Context}$$

It is important that a type $A$ cont assigned to a continuation $\langle\kappa\rangle$ specifies the type of an expression $e$ to fill the hole in $\kappa$, but not the type of the resultant expression $\kappa[\![e]\!]$. For this reason, a continuation is usually said to return an "answer" (of an unknown type) rather than a value of a specific type. For a similar reason, a $\lambda$-abstraction serves as a continuation only if it has a designated return type, *e.g.*, $Ans$, denoting "answers."

The typing rules for the other two constructs respect their reduction rules:

$$\frac{\Gamma, x : A \; \mathsf{cont} \vdash e : A}{\Gamma \vdash \mathsf{callcc}\; x.\, e : A} \; \mathsf{Callcc} \qquad \frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A \; \mathsf{cont}}{\Gamma \vdash \mathsf{throw}\; e_1 \;\mathsf{to}\; e_2 : C} \; \mathsf{Throw}$$

The rule Callcc assigns type $A$ cont to $x$ and type $A$ to $e$ *for the same type* $A$, since if $e$ has type $A$, then the evaluation context when reducing callcc $x.\, e$ also expects a value of type $A$ for the hole in it. callcc $x.\, e$ has the same type as $e$ because, for example, $x$ may not appear in $e$, in which case callcc $x.\, e$ produces the same result as $e$. In the rule Throw, it is safe to assign an arbitrary type $C$ to throw $e_1$ to $e_2$ because its reduction never finishes: there is no value $v$ such that throw $e_1$ to $e_2 \mapsto^* v$. In other words, an "answer" can be an arbitrary type.

An important consequence of the rule Callcc is that a continuation stored in variable $x$ cannot be part of the result of evaluating $e$. For example, callcc $x.\, x$ fails to typecheck because the rule Callcc assigns type $A$ cont to the first $x$ and type $A$ to the second $x$, but there is no way to unify $A$ cont and $A$ (*i.e.*, $A$ cont $\neq A$). Then how can we pass the continuation stored in variable $x$ to the outside of callcc $x.\, e$? Since there is no way to pass it by evaluating $e$, the only hope is to throw it to another continuation! (We will see an example in the next section.)

To complete the definition of the three new constructs, we extend the definition of substitution as follows:

$$\begin{array}{rcll} [e'/x]\mathsf{callcc}\; x.\, e & = & \mathsf{callcc}\; x.\, e & \\ [e'/x]\mathsf{callcc}\; y.\, e & = & \mathsf{callcc}\; y.\, [e'/x]e & \quad \textit{if } x \neq y, y \notin FV(e') \\ [e'/x]\mathsf{throw}\; e_1 \;\mathsf{to}\; e_2 & = & \mathsf{throw}\; [e'/x]e_1 \;\mathsf{to}\; [e'/x]e_2 & \\ [e'/x]\langle\kappa\rangle & = & \langle\kappa\rangle & \end{array}$$

Type safety is stated in the same way as in Theorems **??** and **??**.

## 1.3 Composing two continuations

The goal of this section is to develop a function compose of the following type:

$$\mathsf{compose} : (A \to B) \to B \; \mathsf{cont} \to A \; \mathsf{cont}$$

Roughly speaking, compose $f$ $\langle\kappa\rangle$ joins an incomplete computation (or just a continuation) described by $f$ with $\kappa$ to build a new continuation. To be precise, compose $f$ $\langle\kappa\rangle$ returns a continuation $\kappa'$ such that throwing a value $v$ to $\kappa'$ has the same effect as throwing $f\; v$ to $\kappa$.

**Exercise 1.2.** Give a definition of compose. You have to solve two problems: how to create a correct continuation by placing callcc $x.\, e$ at the right position and how to return the continuation as the return value of compose.

The key observations are:

- throw $v$ to (compose $f$ $\langle\kappa\rangle$) is operationally equivalent to throw $f\; v$ to $\langle\kappa\rangle$.

- For any evaluation context $\kappa'$, both $\mathsf{throw}\ f\ v\ \mathsf{to}\ \langle\kappa\rangle$ and $\kappa'[\![\mathsf{throw}\ f\ v\ \mathsf{to}\ \langle\kappa\rangle]\!]$ evaluate to the same value. More generally, $\langle\mathsf{throw}\ f\ \Box\ \mathsf{to}\ \langle\kappa\rangle\rangle$ and $\langle\kappa'[\![\mathsf{throw}\ f\ \Box\ \mathsf{to}\ \langle\kappa\rangle]\!]\rangle$ are semantically no different.

Thus we define $\mathsf{compose}$ so that $\mathsf{compose}\ f\ \langle\kappa\rangle$ returns $\langle\mathsf{throw}\ f\ \Box\ \mathsf{to}\ \langle\kappa\rangle\rangle$.

First we replace $\Box$ in $\mathsf{throw}\ f\ \Box\ \mathsf{to}\ \langle\kappa\rangle$ by $\mathsf{callcc}\ x.\cdots$ to create a continuation $\langle\kappa'[\![\mathsf{throw}\ f\ \Box\ \mathsf{to}\ \langle\kappa\rangle]\!]\rangle$ (for a certain evaluation context $\kappa'$) which is semantically no different from $\langle\mathsf{throw}\ f\ \Box\ \mathsf{to}\ \langle\kappa\rangle\rangle$:

$$\mathsf{compose} \quad = \quad \lambda f\!:\!A\!\to\!B.\,\lambda k\!:\!B\ \mathsf{cont}.\,\mathsf{throw}\ f\ (\mathsf{callcc}\ x.\cdots)\ \mathsf{to}\ k$$

Then $x$ stores the very continuation that $\mathsf{compose}\ f\ k$ needs to return.

Now how do we return $x$? Obviously $\mathsf{callcc}\ x.\cdots$ cannot return $x$ because $x$ has type $A\ \mathsf{cont}$ while $\cdots$ must have type $A$, which is strictly smaller than $A\ \mathsf{cont}$. Therefore the only way to return $x$ from $\cdots$ is by throwing it to the continuation starting from the hole in $\lambda f\!:\!A\!\to\!B.\,\lambda k\!:\!B\ \mathsf{cont}.\,\Box$:

$$\begin{array}{rcl}\mathsf{compose} & = & \lambda f\!:\!A\!\to\!B.\,\lambda k\!:\!B\ \mathsf{cont}. \\ & & \quad \mathsf{callcc}\ y.\,\mathsf{throw}\ f\ (\mathsf{callcc}\ x.\,\mathsf{throw}\ x\ \mathsf{to}\ y)\ \mathsf{to}\ k\end{array}$$

Note that $y$ has type $A\ \mathsf{cont}\ \mathsf{cont}$. Since $x$ has type $A\ \mathsf{cont}$, $\mathsf{compose}$ ends up throwing $x$ to a continuation which expects another continuation!

## Continuation-passing style