Chapter 1

Exceptions and continuations

In the simply typed λ-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 two extensions to the simply typed λ-calculus for achieving non-local transfers of control. By non-local transfers of control, we mean those reductions that cannot be justified by \(\beta\)-reductions alone. First we briefly consider a primitive form of exception which in its mature form enables us to cope with erroneous conditions such as division by zero, pattern match failure, and array boundary error. Exceptions are also an excellent programming aid. For example, if the specification of a program requires a function \(foo\) that is far from trivial to implement but known to be unused until the late stage of development, we can complete its framework just by declaring \(foo\) such that its body immediately raises an exception.\(^1\) Then we consider continuations which may be thought of as a generalization of evaluation contexts in Chapter ???. The basic idea behind continuations 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 λ-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 1.2) 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.

Both exceptions and continuations are examples of computational effects, called control effects, in that their presence destroys the equivalence between λ-abstractions and mathematical functions. (In comparison, mutable references are often called store effects.) As computational effects do not mix well with the lazy reduction strategy, both kinds of control effects are usually built on top of the eager reduction strategy.\(^2\)

1.1 Exceptions

In order to support exceptions in the simply typed λ-calculus, we introduce two new constructs \(\text{try } e \text{ with } e'\) and \(\text{exn}\):

\[
\text{expression} \quad e \ ::= \ \cdots \mid \text{try } e \text{ with } e' \mid \text{exn}
\]

Informally \(\text{try } e \text{ with } e'\) starts by evaluating \(e\) with an exception handler \(e'\). If \(e\) successfully evaluates to a value \(v\), the whole expression also evaluates to the same value \(v\). In this case, \(e'\) is never visited and is thus ignored. If the evaluation of \(e\) raises an exception by attempting to reduce \(\text{exn}\), the exception

---

\(^1\)The exception Unimplemented in our programming assignments is a good example.

\(^2\)Haskell uses a separate apparatus called monad to deal with computational effects.
handler $e'$ is activated. In this case, the result of evaluating $e'$ serves as the final result of evaluating try $e$ with $e'$. Note that $e'$ may raise another exception, in which case the new exception propagates to the next try $e'_{\text{next}}$ with $e'_{\text{next}}$ such that $e'_{\text{next}}$ encloses try $e$ with $e'$.

Formally the operational semantics is extended with the following reduction rules:

\[
\begin{align*}
\text{exn } e & \mapsto \text{exn } \text{Exn} \\
(\lambda x : A. e) \text{ exn} & \mapsto \text{exn } \text{Exn}' \\
e_1 & \mapsto e'_1 \\
\text{try } e_1 \text{ with } e_2 & \mapsto \text{try } e'_1 \text{ with } e_2 \\
\text{Try} & \mapsto \text{Try}' \\
\text{try } v \text{ with } e & \mapsto \text{try } v \text{ with } e' \\
\text{Try}'' & \mapsto \text{Try}''
\end{align*}
\]

The rules $\text{Exn}$ and $\text{Exn}'$ say that whenever an attempt is made to reduce exn, the whole reduction is canceled and exn starts to propagate. For example, the reduction of $((\lambda x : A. e) \text{ exn}) e'$ eventually ends up with exn:

\[
((\lambda x : A. e) \text{ exn}) e' \mapsto \text{exn } e' \mapsto \text{exn}
\]

In the rule $\text{Try}'$, the reduction bypasses the exception handler $e$ because no exception has been raised. in the rule $\text{Try}''$ the reduction activates the exception handler $e$ because an exception has been raised.

Note that $\text{Exn}$ and $\text{Exn}'$ are two rules specifically designed for propagating exceptions raised within applications. This implies that for all other kinds of constructs, we have to provide separate rules for propagating exceptions. For example, we need the following rule to handle exceptions raised within conditional constructs:

\[
\text{if exn then } e_1 \text{ else } e_2 \mapsto \text{exn } \text{Exn}''
\]

Exercise 1.1. Assuming the eager reduction strategy, give rules for propagating exceptions raised within those constructs for product types and sum types.

1.2 A motivating example for continuations

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:

\[
\begin{align*}
\text{fun multiply } l & = \\
& \text{let} \\
& \quad \text{fun mult } \text{nil} = 1 \\
& \quad | \quad \text{mult } (n :: l') = n * \text{mult } l' \\
& \quad \text{in} \\
& \quad \text{mult } l \\
& \text{end}
\end{align*}
\]

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 remaining elements in the list:

\[
\begin{align*}
\text{fun multiply'} l & = \\
& \text{let} \\
& \quad \text{fun mult } \text{nil} = 1 \\
& \quad | \quad \text{mult } (0 :: l') = 0 \\
& \quad | \quad \text{mult } (n :: l') = n * \text{mult } l' \\
& \quad \text{in} \\
& \quad \text{mult } l \\
& \text{end}
\end{align*}
\]

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 \(\text{mult}\), 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 \(\text{mult}\) altogether after encountering a zero in \(l\), even without returning a zero to previous calls to \(\text{mult}\). What makes this possible is two constructs, callcc and throw, for continuations:

\[
\text{fun multiply'}\ l =
\quad \text{callcc (fn ret =>}
\quad \quad \text{let}
\quad \quad \quad \text{fun mult nil = 1}
\quad \quad \quad | \text{mult (0 :: l')} = \text{throw ret 0}
\quad \quad \quad | \text{mult (n :: l')} = n \times \text{mult l'}
\quad \quad \text{in}
\quad \quad \quad \text{mult l}
\quad \text{end)}
\]

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 \(\text{mult}\) once throw ret 0 is reached.

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

### 1.3 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). That is, by joining a computation with a continuation, we obtain a complete computation. A \(\lambda\)-abstraction \(\lambda x : A. e\) may be seen as a continuation, since it conceptually takes a computation producing a value \(v\) and returns a computation corresponding to \(\left[v/x\right]e\). 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 \(\left[v/x\right]e\). A better example of a continuation is an evaluation context \(\kappa\) which, given an expression \(e\), yields a computation corresponding to \(\kappa \left[\{e\}\right]\). Note that like a \(\lambda\)-abstraction, \(\kappa\) itself does not describe a complete computation. In this chapter, we study evaluation contexts as a means of realizing continuations.

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

\[
\frac{e \rightarrow_\beta e'}{\kappa \left[\{e\}\right] \rightarrow \kappa \left[\{e'/\}ight]} \text{ Red}_\beta
\]

Since the decomposition under the rule \(\text{Red}_\beta\) 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 evaluation context. For example, when a given expression is decomposed into \(\kappa\) and \(e\) by the rule \(\text{Red}_\beta\), 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: \(\langle \kappa \rangle\), callcc \(x. e\), and throw \(e\) 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 to reduce callcc \(x. e\). As a value, \(\langle \kappa \rangle\) is called a continuation.
- callcc \(x. e\) seizes the current evaluation context \(\kappa\) and stores \(\langle \kappa \rangle\) in \(x\) before proceeding to reduce \(e\):

\[
\kappa \left[\{\text{callcc } x. e\}\right] \rightarrow \kappa \left[\{\langle \kappa \rangle / x \right\}^e\right] \text{ Callcc}
\]

In the case that the reduction of \(e\) does not use \(x\) at all, callcc \(x. e\) produces the same result as \(e\).

---

3 In SML/NJ, open the structure SMLofNJ.Cont to test multiply''.
4 Here “prepend” and “prefix” both mean “add to the beginning.”
5 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!
• throw $e$ 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 $\kappa$:

$$\kappa[\text{throw } v \text{ to } (\langle \kappa' \rangle)] \Rightarrow \kappa'[v]$$

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

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

**Exercise 1.2.** What is the result of evaluating each expression below?

1. $\text{fst callcc } x. (\text{true, false})$  \(\Rightarrow^*\) $\text{true}$
2. $\text{fst callcc } x. (\text{true, } \text{throw (false, false) to } x)$  \(\Rightarrow^*\) $\text{false}$
3. $\text{snd callcc } x. (\text{throw (true, true) to } x, \text{false})$  \(\Rightarrow^*\) $\text{true}$

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

1. $\text{fst callcc } x. (\text{true, false})$  \(\Rightarrow^*\) $\text{true}$
2. $\text{fst callcc } x. (e, \text{throw (false, e') to } x)$  \(\Rightarrow^*\) $\text{false}$
3. $\text{snd callcc } x. (\text{throw (e, true) to } x, e')$  \(\Rightarrow^*\) $\text{true}$

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 would we represent its type?) We assign a type $A \text{ cont}$ 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 $A \text{ cont}$:

$$\begin{array}{ll}
\text{type} & A ::= \cdots \mid A \text{ cont} \\
\Gamma \vdash \langle \kappa \rangle : A \text{ cont} \\
\end{array}$$

It is important that a type $A \text{ cont}$ 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., $\text{Ans}$, denoting “answers.”

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

$$\begin{array}{llllllllll}
\Gamma, x : A \text{ cont} \vdash e : A \\
\Gamma \vdash \text{callcc } x. e : A \\
\Gamma \vdash \text{throw } e_1 \text{ to } e_2 : C \\
\end{array}$$

The rule Callcc assigns type $A \text{ cont}$ 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 $e$ at all, 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 $\text{throw } e_1 \text{ to } e_2$.
because its reduction never finishes: there is no value \( v \) such that \( \text{throw} \, e_1 \rightarrow^* \, v \). In other words, an “answer” can be an arbitrary type.

An important consequence of the rule Callcc is that when evaluating callcc \( x \), a continuation stored in variable \( x \) cannot be part of the result of evaluating expression \( e \). For example, callcc \( x \) 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:

\[
\begin{align*}
\ [e'/x]\text{callcc} \, x. \, e &= \quad \text{callcc} \, x. \, e \\
\ [e'/x]\text{callcc} \, y. \, e &= \quad \text{callcc} \, y. \, [e'/x]e \\
\ [e'/x]\text{throw} \, e_1 \rightarrow e_2 &= \quad \text{throw} \, [e'/x]e_1 \rightarrow [e'/x]e_2 \\
\ [e'/x]\langle \kappa \rangle &= \quad \langle \kappa \rangle
\end{align*}
\]

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

### 1.4 Composing two continuations

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

\[
\text{compose} : (A \rightarrow B) \rightarrow B \text{ cont} \rightarrow A \text{ cont}
\]

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 \( v \) to \( \kappa \).

**Exercise 1.3.** 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:

- \( \text{throw} \, v \) to \( \langle \text{compose} \, f \langle \kappa \rangle \rangle \) is operationally equivalent to \( \text{throw} \, f \, v \) to \( \langle \kappa \rangle \).

- For any evaluation context \( \kappa' \), both \( \text{throw} \, f \, v \) to \( \langle \kappa \rangle \) and \( \kappa' \langle \text{throw} \, f \, v \rangle \) evaluate to the same value. More generally, \( \langle \text{throw} \, f \, \square \, v \rangle \) and \( \langle \kappa' \langle \text{throw} \, f \, \square \, v \rangle \rangle \) are semantically no different.

Thus we define compose in such a way that compose \( f \langle \kappa \rangle \) returns \( \langle \text{throw} \, f \, \square \, v \rangle \).

First we replace \( \square \) in \( \langle \text{throw} \, f \, \square \, v \rangle \) by callcc \( x. \, \cdots \) to create a continuation \( \langle \kappa' \langle \text{throw} \, f \, \square \, v \rangle \rangle \) (for a certain evaluation context \( \kappa' \)) which is semantically no different from \( \langle \text{throw} \, f \, \square \, \langle \kappa \rangle \rangle \):

\[
\text{compose} = \quad \lambda f : A \rightarrow B. \, \lambda k : B \text{ cont.} \, \text{throw} \, f \, (\text{callcc} \, x. \, \cdots) \, \text{to} \, k
\]

Then \( x \) stores the very continuation that compose \( f \, k \) needs to return.

Now how do we return \( x \)? Obviously callcc \( x. \, \cdots \) cannot return \( x \) because \( x \) has type \( A \) cont while \( \cdots \) must have type \( A \), which is strictly smaller than \( A \) cont. Therefore the only way to return \( x \) from \( \cdots \) is to throw it to the continuation starting from the hole in \( \lambda f : A \rightarrow B. \, \lambda k : B \text{ cont.} \, \square \):

\[
\text{compose} = \quad \lambda f : A \rightarrow B. \, \lambda k : B \text{ cont.} \\
\text{callcc} \, y. \, \text{throw} \, f \, (\text{callcc} \, x. \, \text{throw} \, \, x \, \text{to} \, y) \, \text{to} \, k
\]

Note that \( y \) has type \( A \) cont cont. Since \( x \) has type \( A \) cont, compose ends up throwing \( x \) to a continuation which expects another continuation!

### 1.5 Exercises

**Exercise 1.4.** Extend the abstract machine \( C \) with new rules for the reduction judgment \( s \rightarrow_c \, s' \) so as to support exceptions. Use a new state \( \sigma \langle\text{exn} \rangle \) to mean that the machine is currently propagating an exception.