Chapter 1

Evaluation contexts

This chapter presents an alternative formulation of the operational semantics for the simply typed λ-calculus. Compared with the operational semantics in Chapter ??, the new formulation is less complex, yet better reflects reductions of expressions in a concrete implementation. The new formulation is a basis for an abstract machine for the simply typed λ-calculus, which, like the Java virtual machine, is capable of running a program independently of the underlying hardware platform.

1.1 Evaluation contexts

Consider the simply typed λ-calculus given in Chapter ??:

| type       | A ::= P | A → A |
| base type  | P ::= bool |
| expression | e ::= x | λx:A. e | e e | true | false | if e then e else e |
| value      | v ::= λx:A. e | true | false |

A reduction judgment $e \mapsto e'$ for the call-by-value strategy is defined inductively by the following reduction rules:

\[
\begin{align*}
\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} & \quad \text{Lam} \\
\frac{e_2 \mapsto e'_2}{(\lambda x:A. e) e_2 \mapsto (\lambda x:A. e) e'_2} & \quad \text{Arg} \\
\frac{e \mapsto e'}{\text{if } e \text{ then } e_1 \text{ else } e_2 \mapsto \text{if } e' \text{ then } e_1 \text{ else } e_2} & \quad \text{If} \\
\frac{\text{if true then } e_1 \text{ else } e_2 \mapsto e_1}{\text{if true then } e_1 \text{ else } e_2 \mapsto e_1} & \quad \text{If}_\text{true} \\
\frac{\text{if false then } e_1 \text{ else } e_2 \mapsto e_2}{\text{if false then } e_1 \text{ else } e_2 \mapsto e_2} & \quad \text{If}_\text{false}
\end{align*}
\]

Since only the rules $\text{App}$, $\text{If}_\text{true}$, and $\text{If}_\text{false}$ have no premise, every derivation tree for a reduction judgment $e \mapsto e'$ must end with an application of one of these rules:

\[
\begin{align*}
\frac{\text{if true then } e_1 \text{ else } e_2 \mapsto e_1}{e \mapsto e'} & \quad \text{If}_\text{true} \\
\frac{\text{if false then } e_1 \text{ else } e_2 \mapsto e_2}{e \mapsto e'} & \quad \text{If}_\text{false}
\end{align*}
\]
Thus the reduction of an expression \( e \) amounts to locating an appropriate subexpression \( (\lambda x : A. e''') v \), if true then \( e_1 \) else \( e_2 \), or if false then \( e_1 \) else \( e_2 \) of \( e \) and applying a corresponding reduction rule.

As an example, let us reduce the following expression:

\[
 e = (if \ (\lambda x : A. e''') v \ then \ e_1 \ else \ e_2) \ e' 
\]

The reduction of \( e \) cannot proceed without first reducing the underlined subexpression \( (\lambda x : A. e''') v \) by the rule \( \text{App} \), as shown in the following derivation tree:

\[
\begin{array}{c}
(\lambda x : A. e''') v \rightarrow [v/x]e''' \\
\text{App} \\
\end{array} \\
\begin{array}{c}
\frac{if \ (\lambda x : A. e''') v \ then \ e_1 \ else \ e_2 \rightarrow \cdots}{Lam} \\
\text{If} \\
\end{array} \\
\]

Then we may think of \( e \) as consisting of two parts: a subexpression, or a redex, \( (\lambda x : A. e''') v \) which actually reduces to another expression \([v/x]e'''\) by the rule \( \text{App} \), and the rest which remains intact during the reduction. Note that the second part is not an expression because it is obtained by erasing the redex from \( e \). We write the second part as \((if \ □ \ then \ e_1 \ else \ e_2) \ e'\) where the hole \( □ \) indicates the position of the redex.

We refer to an expression with a hole in it, such as \((if \ □ \ then \ e_1 \ else \ e_2) \ e'\), as an evaluation context. The hole indicates the position of the redex (to be reduced by one of the rules \( \text{App} \), \( \text{If}_\text{true} \), and \( \text{If}_\text{false} \)) for the next step. Note that we may not use the rule \( \text{Lam} \), \( \text{Arg} \), or \( \text{If} \) to reduce the redex, since none of these rules reduces the whole redex in a single step.

Since the hole in an evaluation context indicates the position of a redex, every expression is decomposed into a unique evaluation context and a unique redex under a particular reduction strategy. For the same reason, not every expression with a hole in it is a valid evaluation context. For example, \((e_1 \ e_2) \ □ \) is not a valid evaluation context under the call-by-value strategy because given an expression \((e_1 \ e_2) \ e'\), we have to reduce \( e_1 \) \( e_2 \) before we reduce \( e' \). These two observations show that a particular reduction strategy specifies a unique inductive definition of evaluation contexts. The call-by-value strategy results in the following definition:

\[
\begin{align*}
\text{evaluation context} & \quad \kappa \ ::= \ □ \ | \ \kappa \ e \ | \ (\lambda x : A. \ e) \ \kappa \ | \ \text{if} \ \kappa \ \text{then} \ e \ \text{else} \ e \\
\end{align*}
\]

\( \kappa \ e \) is an evaluation context for \( e' \ e \) where \( e' \) needs to be further reduced; \( (\lambda x : A. \ e) \ \kappa \) is an evaluation context for \( (\lambda x : A. \ e) \ e' \) where \( e' \) needs to be further reduced. Similarly if \( \kappa \) then \( e_1 \) else \( e_2 \) is an evaluation context for if \( e' \) then \( e_1 \) else \( e_2 \) where \( e' \) needs to be further reduced.

Let us write \( \kappa[e] \) for an expression obtained by filling the hole in \( \kappa \) with \( e \). Here are a few examples:

\[
\begin{align*}
\ □ \ [((\lambda x : A. e''') v)] & = (\lambda x : A. e''') v \\
(if \ □ \ then \ e_1 \ else \ e_2) \ [((\lambda x : A. e''') v)] & = (if \ (\lambda x : A. e''') v \ then \ e_1 \ else \ e_2) \\
((if \ □ \ then \ e_1 \ else \ e_2) \ e') \ [((\lambda x : A. e''') v)] & = (if \ (\lambda x : A. e''') v \ then \ e_1 \ else \ e_2) \ e' \\
\end{align*}
\]

A formal definition of \( \kappa[e] \) is given as follows:

\[
\begin{align*}
\ □ \ [e] & = e \\
(\kappa e') \ [e] & = \kappa[e] \ e' \\
((\lambda x : A. e') \ \kappa) \ [e] & = (\lambda x : A. e') \ \kappa[e] \\
(if \ \kappa \ \text{then} \ e_1 \ \text{else} \ e_2) \ [e] & = \text{if} \ \kappa[e] \ \text{then} \ e_1 \ \text{else} \ e_2 \\
\end{align*}
\]

Now consider an expression which is known to reduce to another expression. We can write it as \( \kappa[e] \) for a unique evaluation context \( \kappa \) and a unique redex \( e \). Since \( \kappa[e] \) is known to reduce to another expression, \( e \) must also reduce to another expression \( e' \). We write \( e \rightarrow_\beta e' \) to indicate that the reduction of \( e \) to \( e' \) uses the rule \( \text{App} \), \( \text{If}_\text{true} \), or \( \text{If}_\text{false} \). Then the following reduction rule alone is enough to completely specify a reduction strategy because the order of reduction is implicitly determined by the definition of evaluation contexts:

\[
\begin{array}{c}
e \rightarrow_\beta e' \\
\kappa[e] \rightarrow \kappa[e'] \\
\text{Red}_\beta \\
\end{array}
\]
The reduction relation $\rightsquigarrow$ is defined by the following equations:

\[
\begin{align*}
(\lambda x : A. e) v &\rightsquigarrow [v/x]e \\
\text{if true then } e_1 \text{ else } e_2 &\rightsquigarrow e_1 \\
\text{if false then } e_1 \text{ else } e_2 &\rightsquigarrow e_2 \\
\end{align*}
\]

\[
\frac{e \rightsquigarrow e'}{\kappa[e] \rightsquigarrow \kappa[e']} \quad \text{Red}_\beta
\]

Figure 1.1: Call-by-value operational semantics using evaluation contexts

The reduction relation $\rightsquigarrow$ is defined by the following equations:

\[
\begin{align*}
(\lambda x : A. e) e' &\rightsquigarrow [e'/x]e \\
\text{if true then } e_1 \text{ else } e_2 &\rightsquigarrow e_1 \\
\text{if false then } e_1 \text{ else } e_2 &\rightsquigarrow e_2 \\
\end{align*}
\]

\[
\frac{e \rightsquigarrow e'}{\kappa[e] \rightsquigarrow \kappa[e']} \quad \text{Red}_\beta
\]

Figure 1.2: Call-by-name operational semantics using evaluation contexts

Figure 1.1 summarizes how to use evaluation contexts to specify the call-by-value operational semantics. An example of a reduction sequence is shown below. In each step, we underline the redex and show how to decompose a given expression into a unique evaluation context and a unique redex.

\[
\begin{align*}
(\text{if } (\text{if } x : \text{bool} \text{ then } \lambda y : \text{bool}. y \text{ else } \lambda z : \text{bool}. z) \text{ true }) &\rightsquigarrow (\text{if } \text{false }) [\text{if true then } \lambda y : \text{bool}. y \text{ else } \lambda z : \text{bool}. z] \\
&\rightsquigarrow (\text{false }) [\lambda y : \text{bool}. y] \\
&\rightsquigarrow (\text{false }) [\text{true}] \\
&\rightsquigarrow \text{true} \\
\end{align*}
\]

\[
\begin{align*}
(\text{if } (\text{if } x : \text{bool} \text{ then } \lambda y : \text{bool}. y \text{ else } \lambda z : \text{bool}. z) \text{ true }) &\rightsquigarrow (\text{if } \text{false }) [\text{if true then } \lambda y : \text{bool}. y \text{ else } \lambda z : \text{bool}. z] \\
&\rightsquigarrow (\text{false }) [\lambda y : \text{bool}. y] \\
&\rightsquigarrow (\text{false }) [\text{true}] \\
&\rightsquigarrow \text{true} \\
\end{align*}
\]

In order to obtain the call-by-name operational semantics, we only have to change the inductive definition of evaluation contexts and the reduction relation $\rightsquigarrow$, as shown in Figure 1.2. With a proper understanding of evaluation contexts, it should also be straightforward to incorporate those reduction rules in Chapter ?? into the definition of evaluation contexts and the reduction relation $\rightsquigarrow$. The reader is encouraged to try to augment the definition of evaluation contexts and the reduction relation $\rightsquigarrow$. See Figures 1.3 and 1.4 for the result.

**Exercise 1.1.** Give a definition of evaluation contexts corresponding to the weird reduction strategy specified in Exercise ??.

### 1.2 Type safety

As usual, type safety consists of progress and type preservation:
We write
\[ \text{Theorem 1.3 (Type preservation).} \]
\[ \text{Theorem 1.2 (Progress).} \]
\[ \text{Lemma 1.5.} \]
\[ \text{Proposition 1.4.} \]

4 May 17, 2011

By using the definition of the rules \( \Box \text{ctx} \), \( \text{Lam}_{\text{ctx}} \), \( \text{Arg}_{\text{ctx}} \), and \( \text{If}_{\text{ctx}} \) are admissible.

**Proposition 1.4.** The rules \( \Box_{\text{ctx}} \), \( \text{Lam}_{\text{ctx}} \), \( \text{Arg}_{\text{ctx}} \), and \( \text{If}_{\text{ctx}} \) are admissible.

**Proof.** By using the definition of the rules \( \Gamma \vdash \kappa : A \Rightarrow C \). We show the case for the rule \( \text{Lam}_{\text{ctx}} \).

\[
\begin{align*}
\text{Case} & \quad \Gamma \vdash \kappa : A \Rightarrow B \Rightarrow C & \Gamma \vdash e : B \\
\Gamma \vdash \kappa & : A \Rightarrow B \Rightarrow C & \Gamma \vdash \kappa e : A \Rightarrow C
\end{align*}
\]

assumptions

The proof of **Theorem 1.2** is similar to the proof of Theorem ?? and Theorem 1.3 uses the following lemma whose proof uses Lemma ??:

**Lemma 1.5.** If \( \Gamma \vdash \kappa[e]: C \), then \( \Gamma \vdash e : A \) and \( \Gamma \vdash \kappa : A \Rightarrow C \) for some type \( A \).
Proof. By structural induction on \( \kappa \). We show the case for \( \kappa = \kappa' \ e' \).

Case \( \kappa = \kappa' \ e' \)
\[
\begin{align*}
\Gamma \vdash \kappa[e] : C & \quad \text{assumption} \\
\Gamma \vdash (\kappa'[\e]) e' : C & \quad \kappa[e] = (\kappa'[\e]) e' \\
\Gamma \vdash \kappa'[\e] : B \rightarrow C \text{ and } \Gamma \vdash e' : B \text{ for some type } B & \quad \text{by Lemma ??} \\
\Gamma \vdash e : A \text{ and } \Gamma \vdash \kappa' : A \rightarrow B \rightarrow C \text{ for some type } A & \quad \text{by induction hypothesis on } \kappa' \\
\Gamma \vdash \kappa : A \Rightarrow C & \quad \text{by the rule } \text{Lam}_{\text{ctx}} \\
\end{align*}
\]

1.3 Abstract machine C

The concept of evaluation context leads to a concise formulation of the operational semantics, but it is not suitable for an actual implementation of the simply typed \( \lambda \)-calculus. The main reason is that the rule \textit{Red}_\beta tacitly assumes an automatic decomposition of a given expression into a unique evaluation context and a unique redex, but it may in fact require an explicit analysis of the given expression in several steps. For example, in order to rewrite

\[
e = (\text{if } (\lambda x: A. e'') v \text{ then } e_1 \text{ else } e_2) e'
\]

as \(((\text{if } \square \text{ then } e_1 \text{ else } e_2) e')[(\lambda x: A. e'') v] \), we would analyze \( e \) in several steps:

\[
\begin{align*}
e & = (\text{if } (\lambda x: A. e'') v \text{ then } e_1 \text{ else } e_2) e' \\
& = (\square e')[(\lambda x: A. e'') v \text{ then } e_1 \text{ else } e_2] \\
& = (\text{if } \square \text{ then } e_1 \text{ else } e_2) e'[(\lambda x: A. e'') v] \\
\end{align*}
\]

The abstract machine \( C \) is another formulation of the operational semantics in which such an analysis is explicit.

Roughly speaking, the abstract machine \( C \) replaces an evaluation context by a stack of frames such that each frame corresponds to a specific step in the analysis of a given expression:

\[
\begin{align*}
\text{frame} & \quad \phi \ ::= \square e \ | \ (\lambda x: A. e) \square \ | \ \text{if } \square \text{ then } e_1 \text{ else } e_2 \\
\text{stack} & \quad \sigma \ ::= \square \ | \ \sigma ; \phi
\end{align*}
\]

Frames are special cases of evaluation contexts which are not defined inductively. Thus we may write \( \phi[e] \) for an expression obtained by filling the hole in \( \phi \) with \( e \). A stack of frames also represents an evaluation context in that given an expression, it determines a unique expression. To be specific, a stack \( \sigma \) and an expression \( e \) determine a unique expression \( \sigma[e] \) defined inductively as follows:

\[
\begin{align*}
\square[e] & = e \\
(\sigma ; \phi)[e] & = \sigma[\phi[e]]
\end{align*}
\]

If we write \( \sigma \) as \( \square ; \phi_1 ; \phi_2 ; \cdots ; \phi_n \) for \( n \geq 0 \), \( \sigma[e] \) may be written as

\[
\square[\phi_1[\phi_2[\cdots[\phi_n[e]]\cdots]]].
\]

Now, for example, an implicit analysis of \( e = (\text{if } (\lambda x: A. e'') v \text{ then } e_1 \text{ else } e_2) e' \) shown above can be made explicit by using a stack of frames:

\[
e = (\square ; \square e' ; \text{if } \square \text{ then } e_1 \text{ else } e_2)[(\lambda x: A. e'') v]
\]

Note that the top frame of a stack \( \sigma ; \phi \) is \( \phi \) and that the bottom of a stack is always \( \square \).

A state of the abstract machine \( C \) is specified by a stack \( \sigma \) and an expression \( e \), in which case the machine can be thought of as reducing an expression \( \sigma[e] \). In addition, the state includes a flag to indicate whether \( e \) needs to be further analyzed or has already been reduced to a value. Thus we use the following definition of states:

\[
\text{state} \quad s ::= \sigma \triangleright e \triangleright \sigma \triangleright v
\]

May 17, 2011
• \( \sigma \triangleright e \) means that the machine is currently reducing \( \sigma[e] \), but has yet to analyze \( e \).

• \( \sigma \triangleright v \) means that the machine is currently reducing \( \sigma[v] \) and has already analyzed \( v \). That is, it is returning \( v \) to the top frame of \( \sigma \).

Thus, if an expression \( e \) evaluates to a value \( v \), a state \( \sigma \triangleright e \) will eventually lead to another state \( \sigma \triangleright v \). As a special case, the initial state of the machine evaluating \( e \) is always \( \square \triangleright e \) and the final state \( \square \triangleright v \) if \( e \) evaluates to \( v \).

A state transition in the abstract machine \( C \) is specified by a reduction judgment \( s \xrightarrow{C} s' \); we write \( s \xrightarrow{C} e \) for the reflexive and transitive closure of \( s \xrightarrow{C} \). The guiding principle for state transitions is to maintain the invariant that \( e \xrightarrow{*} v \) holds if and only if \( \sigma \triangleright e \xrightarrow{C} \sigma \triangleright v \) holds for any stack \( \sigma \). The rules for the reduction judgment \( s \xrightarrow{C} s' \) are as follows:

\[
\begin{align*}
\sigma \triangleright v \xrightarrow{C} \sigma \triangleright v & \quad \text{Val}_C \\
\sigma \triangleright e_1 e_2 \xrightarrow{C} \sigma \triangleright e_1 & \quad \text{Lam}_C \\
\sigma \triangleright (\lambda x : A. e) \xrightarrow{C} \sigma \triangleright e & \quad \text{Arg}_C \\
\sigma \triangleright (\lambda x : A. e) v \xrightarrow{C} \sigma \triangleright [v/x]e & \quad \text{App}_C \\
\sigma \triangleright \text{if } e \text{ then } e_1 \text{ else } e_2 \xrightarrow{C} \sigma \triangleright e_1 & \quad \text{If}_C \\
\sigma \triangleright \text{if } e \text{ then } e_1 \text{ else } e_2 \xrightarrow{C} \sigma \triangleright e_2 & \quad \text{If}_{\text{false}}_C
\end{align*}
\]

An example of a reduction sequence is shown below. Note that it begins with a state \( \square \triangleright e \) and ends with a state \( \square \triangleright v \).

\[
\begin{align*}
\square \triangleright (\lambda x : \text{bool}. \, x) \text{ true } \lambda y : \text{bool}. \, y \text{ else } \lambda z : \text{bool}. \, z \text{ true} & \quad \text{Lam}_C \\
\square \triangleright (\lambda y : \text{bool}. \, y) \text{ true } \lambda y : \text{bool}. \, y \text{ else } \lambda z : \text{bool}. \, z \text{ true} & \quad \text{If}_C \\
\square \triangleright (\lambda y : \text{bool}. \, y) \text{ true } \lambda y : \text{bool}. \, y \text{ else } \lambda z : \text{bool}. \, z \text{ true} & \quad \text{Lam}_C \\
\square \triangleright (\lambda y : \text{bool}. \, y) \text{ true } \lambda y : \text{bool}. \, y \text{ else } \lambda z : \text{bool}. \, z \text{ true} & \quad \text{If}_C \\
\square \triangleright (\lambda y : \text{bool}. \, y) \text{ true } \lambda y : \text{bool}. \, y \text{ else } \lambda z : \text{bool}. \, z \text{ true} & \quad \text{Lam}_C \\
\square \triangleright (\lambda y : \text{bool}. \, y) \text{ true } \lambda y : \text{bool}. \, y \text{ else } \lambda z : \text{bool}. \, z \text{ true} & \quad \text{If}_C \\
\square \triangleright (\lambda y : \text{bool}. \, y) \text{ true } \lambda y : \text{bool}. \, y \text{ else } \lambda z : \text{bool}. \, z \text{ true} & \quad \text{Lam}_C \\
\square \triangleright (\lambda y : \text{bool}. \, y) \text{ true } \lambda y : \text{bool}. \, y \text{ else } \lambda z : \text{bool}. \, z \text{ true} & \quad \text{If}_C \\
\square \triangleright (\lambda y : \text{bool}. \, y) \text{ true } \lambda y : \text{bool}. \, y \text{ else } \lambda z : \text{bool}. \, z \text{ true} & \quad \text{Lam}_C \\
\square \triangleright (\lambda y : \text{bool}. \, y) \text{ true } \lambda y : \text{bool}. \, y \text{ else } \lambda z : \text{bool}. \, z \text{ true} & \quad \text{If}_C \\
\square \triangleright \lambda y : \text{bool}. \, y & \quad \text{Arg}_C \\
\square \triangleright \lambda y : \text{bool}. \, y & \quad \text{Arg}_C \\
\square \triangleright (\lambda y : \text{bool}. \, y) \text{ true } & \quad \text{Lam}_C \\
\square \triangleright (\lambda y : \text{bool}. \, y) \text{ true } & \quad \text{Lam}_C
\end{align*}
\]

1.4 Correctness of the abstract machine \( C \)

This section presents a proof of the correctness of the abstract machine \( C \) as stated in the following theorem:

**Theorem 1.6.** \( e \xrightarrow{*} v \) if and only if \( \square \triangleright e \xrightarrow{C} \square \triangleright v \).

A more general version of the theorem allows any stack \( \sigma \) in place of \( \square \), but we do not prove it here. For the sake of simplicity, we also do not consider expressions of type \( \text{bool} \) altogether.

It is a good and challenging exercise to prove the theorem. The main difficulty lies in finding several lemmas necessary for proving the theorem, not in constructing their proofs. The reader is encouraged to guess these lemmas without having to write their proofs.

The proof uses a generalization of \( \kappa[\cdot] \) and \( \sigma[\cdot] \) over evaluation contexts:

\[
\begin{align*}
\square[\kappa'] & = \kappa' \\
(\kappa e)[\kappa'] & = \kappa[\kappa'] e \\
((\lambda x : A. e) \kappa)[\kappa'] & = (\lambda x : A. e) \kappa[\kappa']
\end{align*}
\]

May 17, 2011
Note that \( \kappa[\kappa'] \) and \( \sigma[\kappa] \) are evaluation contexts.

**Proposition 1.7.** \( \kappa[\kappa'[e]] = \kappa[\kappa'][e] \).

**Proof.** By structural induction on \( \kappa \). We show two cases.

Case \( \kappa = \Box \):
\[
\Box[\kappa'[e]] = \kappa'[e] = \Box[\kappa'][e]
\]

Case \( \kappa = \kappa'' e'' \):
\[
(k'' e'')[\kappa'[e]] = \kappa''[\kappa'[e]] e'' = \kappa''[\kappa'][e] e'' = (\kappa''[\kappa'][e''])[e] = (\kappa''[e''])[\kappa'][e]
\]

**Proposition 1.8.** \( \sigma[\kappa[e]] = \sigma[\kappa][e] \).

**Proof.** By structural induction on \( \kappa \). The second case uses Proposition 1.7.

Case \( \sigma = \Box \):
\[
\Box[\kappa[e]] = \kappa[e] = \Box[\kappa][e]
\]

Case \( \sigma = \sigma'; \phi \):
\[
(\sigma'; \phi)[\kappa[e]] = \sigma'[\phi[\kappa[e]]] = \sigma'[\phi[\kappa]][e] = (\sigma'; \phi)[\kappa][e]
\]

**Lemma 1.9.** For \( \sigma \) and \( \kappa \), there exists \( \sigma' \) such that \( \sigma \triangleright \kappa[e] \rightarrow^\ast \sigma' \triangleright e \) and \( \sigma[\kappa] = \sigma'[\Box] \) for any expression \( e \).

**Proof.** By structural induction on \( \kappa \). We show two cases.

Case \( \kappa = \Box \):
We let \( \sigma' = \sigma \).

Case \( \kappa = \kappa' e' \):
\[
\sigma \triangleright (\kappa' e')[e] = \sigma \triangleright \kappa'[e] e' \rightarrow^\ast \sigma; \Box e' \triangleright \kappa'[e]
\]
\[
\sigma; \Box e' \triangleright \kappa'[e] \rightarrow^\ast \sigma' \triangleright e \text{ and } (\sigma; \Box e')[\kappa'] = \sigma'[\Box]
\]
\[
\sigma[\kappa] = \sigma[\kappa'[e']] = \sigma[(\Box e')[\kappa']] = (\sigma; \Box e')[\kappa'] = \sigma'[\Box]
\]

**Lemma 1.10.** Suppose \( \sigma[e] = \kappa[f v] \) where \( f \) is a \( \lambda \)-abstraction. Then one of the following cases holds:

1. \( \sigma \triangleright e \rightarrow^\ast \sigma' \triangleright \kappa'[f v] \) and \( \sigma'[\kappa'] = \kappa \)
2. \( \sigma = \sigma'; \Box v \) and \( e = f \) and \( \sigma'[\Box] = \kappa \)
3. \( \sigma = \sigma'; f \Box \) and \( e = v \) and \( \sigma'[\Box] = \kappa \)

**Proof.** By structural induction on \( \sigma \). We show two cases.

Case \( \sigma = \Box \):
\[
\sigma[e] = e = \kappa[f v]
\]
assumption
\[
\sigma \triangleright e = \Box \triangleright \kappa[f v]
\]
(1) \( \sigma \triangleright e \rightarrow^\ast \sigma; \Box \triangleright \kappa[f v] \) and \( \Box[\kappa] = \kappa \)

Case \( \sigma = \sigma'; \Box e' \):
\[
\sigma[e] = (\sigma'; \Box e')[e] = \sigma'[(\Box e')[e]] = \sigma'[e e'] = \kappa[f v]
\]

Subcase (1) \( \sigma' \triangleright e e' \rightarrow^\ast \sigma'' \triangleright \kappa'[f v] \) and \( \sigma''[\kappa'] = \kappa \):
by induction hypothesis on \( \sigma' \)
\[
\sigma' \triangleright e e' \rightarrow^\ast \sigma'' \triangleright \kappa'[f v]
\]
assumption
(1) \( \sigma \triangleright e \rightarrow^\ast \sigma'' \triangleright \kappa'[f v] \) and \( \sigma''[\kappa'] = \kappa \)
\[
\sigma' = \sigma'' \text{ and } e e' = \kappa'[f v], \text{ and } \kappa' = \Box
\]
assumption
\[
e = f \text{ and } e' = v
\]
(2) \( \sigma = \sigma'; \Box v \) and \( e = f \) and \( \sigma'[\Box] = \sigma''[\kappa'] = \kappa \)

May 17, 2011 7
Corollary 1.12. \(\sigma' = \sigma''\) and \(e e' = \kappa'[f v]\), and \(\kappa' = \kappa'' e'\)

\(e = \kappa''[f v]\)

(1) \[\sigma \mapsto e \mapsto^* \sigma \mapsto e' \mapsto^* \sigma''[f v]\]

\(\sigma' = \sigma''\) and \(e e' = \kappa'[f v]\), and \(\kappa' = e \kappa''\)

e is a \(\lambda\)-abstraction and \(e' = \kappa''[f v]\)

(1) \[\sigma \mapsto e = \sigma'; \quad \square v \mapsto e \mapsto \sigma' \mapsto e' = \sigma' e \square \mapsto \kappa''[f v]\]

and \((\sigma'; e \square)[\kappa''] = \sigma'[e \kappa''] = \sigma''[\kappa''] = \kappa\)

Proof. By Lemma 1.10, we need to consider the following three cases:

(1) \[\sigma \mapsto e \mapsto^* \sigma' \mapsto \kappa'[f v] \quad \text{where} \quad \sigma'[\kappa''] = \sigma''[\square] \quad \text{by Lemma 1.9}\]

We let \(\sigma^* = \sigma''\).

(2) \[\sigma = \sigma'; \quad \square v \quad \text{and} \quad e = f \quad \text{and} \quad \sigma'[\square] = \kappa\]

\[\sigma \mapsto e = \sigma'; \quad \square v \quad \mapsto f \mapsto \sigma' \mapsto e' \quad \text{by Proposition 1.8}\]

We let \(\sigma^* = \sigma'\).

(3) \[\sigma = \sigma'; \quad f \quad \square \quad \text{and} \quad e = v \quad \text{and} \quad \sigma'[\square] = \kappa\]

\[\sigma \mapsto e = \sigma'; \quad f \quad \square \quad \mapsto v \mapsto \sigma' \mapsto e' \quad \text{by Proposition 1.8}\]

We let \(\sigma^* = \sigma'\).

Corollary 1.12. Suppose \(e_1 \mapsto e_2 \quad \text{and} \quad \sigma[e] = e_1\). Then there exist \(\sigma'\) and \(e'\) such that \(\sigma \mapsto e \mapsto \sigma' \mapsto e' \quad \text{and} \quad \sigma'[e'] = e_2\).

We leave it to the reader to prove all results given below.

Proposition 1.13. Suppose \(e \mapsto^* v\) and \(\sigma[e'] = e\). Then \(\sigma \mapsto e' \mapsto \square v\).

Corollary 1.14. If \(e \mapsto^* v\), then \(\square \mapsto e \mapsto \square v\).

Proposition 1.15.

If \(\sigma \mapsto e \mapsto \sigma' \mapsto e'\), then \(\sigma[e] \mapsto^* \sigma'[e']\).

If \(\sigma \mapsto e \mapsto \sigma' \mapsto \square v',\) then \(\sigma[e] \mapsto^* \sigma'[v']\).

Corollary 1.16.

If \(\sigma \mapsto e \mapsto \sigma' \mapsto e'\), then \(\sigma[e] \mapsto^* \sigma'[e']\).

If \(\sigma \mapsto e \mapsto \sigma' \mapsto \square v',\) then \(\sigma[e] \mapsto^* \sigma'[v']\).

Corollary 1.17. If \(\square \mapsto e \mapsto \square v,\) then \(e \mapsto^* v\).

Corollaries 1.14 and 1.17 prove Theorem 1.6
1.5 Safety of the abstract machine C

The safety of the abstract machine C is proven independently of its correctness. We use two judgments to describe the state of C with three inference rules given below:

- \( s \) okay means that \( s \) is an “okay” state. That is, C is ready to analyze a given expression.
- \( s \) stop means that \( s \) is a “stop” state. That is, C has finished reducing a given expression.

\[
\begin{align*}
\sigma[\Box] : A \rightarrow C & \quad \vdash e : A \quad \text{Okay} \quad \sigma[\Box] : A \rightarrow C & \quad \vdash v : A \quad \text{Okay} \quad \Box \quad \vdash v \quad \text{Stop}
\end{align*}
\]

The first clause in the following theorem may be thought of as the progress property of the abstract machine C; the second clause may be thought of as the “state” preservation property.

**Theorem 1.18 (Safety of the abstract machine C).**

If \( s \) okay, then either \( s \) stop or there exists \( s' \) such that \( s \rightarrow_C s' \).

If \( s \) okay and \( s \rightarrow_C s' \), then \( s' \) okay.

1.6 Exercises

**Exercise 1.19.** Prove Theorems 1.2 and 1.3.

**Exercise 1.20.** Consider the simply typed \( \lambda \)-calculus extended with product types, sum types, and the fixed point construct.

expression \( e ::= x | \lambda x : A . e | e\ e | (e, e) | \text{fst } e | \text{snd } e | () | \text{inl}_A e | \text{inr}_A e | \text{case } e \text{ of inl } x . e | \text{inr } x . e | \text{fix } x : A . e | \text{true } | \text{false } | \text{if } e \text{ then } e \text{ else } e \) 

value \( v ::= \lambda x : A . e | (v, v) | () | \text{inl}_A v | \text{inr}_A v | \text{true } | \text{false } \)

Assuming the call-by-value strategy, extend the definition of frames and give additional rules for the reduction judgment \( s \rightarrow_C s' \) for the abstract machine C. See Figure 1.5 for an answer.

**Exercise 1.21.** Prove Theorem 1.18.
frame $\phi ::= \square | v \square | (\square, e) | (v, \square) | \text{fst} \square | \text{snd} \square |
\begin{align*}
\text{inl}_A \square \vdash \text{inr}_A \square \vdash \text{case} \square \text{of inl} x. e \mid \text{inr} x. e \mid \text{if} \square \text{then} e_1 \text{else} e_2
\end{align*}

\begin{align*}
\sigma \triangleright (e_1, e_2) & \mapsto_\mathcal{C} \sigma; (\square, e_2) \triangleright e_1 & \text{Pair}_\mathcal{C} \\
\sigma; (v_1, \square) & \triangleright (v_2, v_2) \mapsto_\mathcal{C} \sigma \triangleright (v_1, v_2) & \text{Pair}'_\mathcal{C} \\
\sigma \triangleright \text{fst} e & \mapsto_\mathcal{C} \sigma; \text{fst} \square \triangleright e & \text{Fst}_\mathcal{C} \\
\sigma \triangleright \text{snd} e & \mapsto_\mathcal{C} \sigma; \text{snd} \square \triangleright e & \text{Snd}_\mathcal{C} \\
\sigma \triangleright \text{inl}_A e & \mapsto_\mathcal{C} \sigma; \text{inl}_A \square \triangleright e & \text{Inl}_\mathcal{C} \\
\sigma \triangleright \text{inr}_A e & \mapsto_\mathcal{C} \sigma; \text{inr}_A \square \triangleright e & \text{Inr}_\mathcal{C} \\
\sigma \triangleright \text{case} e \text{ of inl} x_1. e_1 \mid \text{inr} x_2. e_2 \mapsto_\mathcal{C} \sigma; \text{case} \square \text{ of inl} x_1. e_1 \mid \text{inr} x_2. e_2 \triangleright e & \text{Case}_\mathcal{C} \\
\sigma; \text{case} \square \text{ of inl} x_1. e_1 \mid \text{inr} x_2. e_2 & \triangleleft \text{inl}_A v \mapsto_\mathcal{C} \sigma \triangleright [v/x_1] e_1 & \text{Case}'_\mathcal{C} \\
\sigma; \text{case} \square \text{ of inl} x_1. e_1 \mid \text{inr} x_2. e_2 & \triangleleft \text{inr}_A v \mapsto_\mathcal{C} \sigma \triangleright [v/x_2] e_2 & \text{Case}''_\mathcal{C} \\
\sigma \triangleright \text{fix} x: A. e & \mapsto_\mathcal{C} \sigma \triangleright [\text{fix} x: A. e/x] e & \text{Fix}_\mathcal{C}
\end{align*}

Figure 1.5: Abstract machine C for product types, sum types, and the fixed point construct