# Chapter 1

# Evaluation contexts

This chapter presents an alternative formulation of the operational semantics for the simply typed $\lambda$-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 $\lambda$-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 $\lambda$-calculus given in Chapter **??**:

$$\begin{array}{rrcl} \text{type} & A & ::= & P \mid A \to A \\ \text{base type} & P & ::= & \mathsf{bool} \\ \text{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\ \text{value} & v & ::= & \lambda x\!:\!A.\, e \mid \mathsf{true} \mid \mathsf{false} \end{array}$$

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

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \mathit{Lam} \qquad \frac{e_2 \mapsto e_2'}{(\lambda x\!:\!A.\, e)\ e_2 \mapsto (\lambda x\!:\!A.\, e)\ e_2'}\ \mathit{Arg} \qquad \frac{}{(\lambda x\!:\!A.\, e)\ v \mapsto [v/x]e}\ \mathit{App}$$

$$\frac{e \mapsto e'}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}\ \mathit{If}$$

$$\frac{}{\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_1}\ \mathit{If}_{\mathsf{true}} \qquad \frac{}{\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_2}\ \mathit{If}_{\mathsf{false}}$$

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

$$\dfrac{\dfrac{}{(\lambda x\!:\!A.\, e'')\ v \mapsto [v/x]e''}\ \mathit{App}}{\vdots}{e \mapsto e'}$$

$$\dfrac{\dfrac{}{\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_1}\ \mathit{If}_{\mathsf{true}}}{\vdots}{e \mapsto e'} \qquad \dfrac{\dfrac{}{\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_2}\ \mathit{If}_{\mathsf{false}}}{\vdots}{e \mapsto e'}$$

Thus the reduction of an expression $e$ amounts to locating an appropriate subexpression $(\lambda x\!:\!A.\, e'')\ v$, $\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2$, or $\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2$ of $e$ and applying a corresponding reduction rule.

As an example, let us reduce the following expression:

$$e \quad = \quad (\mathsf{if}\ \underline{(\lambda x\!:\!A.\, e'')\ v}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ e'$$

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

$$\dfrac{\dfrac{\dfrac{}{(\lambda x\!:\!A.\, e'')\ v \mapsto [v/x]e''}\ App}{\mathsf{if}\ (\lambda x\!:\!A.\, e'')\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto \cdots}\ If}{(\mathsf{if}\ (\lambda x\!:\!A.\, e'')\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ e' \mapsto \cdots}\ Lam$$

Then we may think of $e$ as consisting of two parts: a subexpression $(\lambda x\!:\!A.\, e'')\ v$ which actually reduces to another expression $[v/x]e''$ by the rule $App$, 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 first part from $e$. We write it as $(\mathsf{if}\ \Box\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ e'$ where the hole $\Box$ indicates the position of the subexpression that has been erased, or equivalently, the subexpression to be reduced in the next step.

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

Since the hole in an evaluation context indicates the position of a subexpression to be reduced in the next step, every expression is decomposed into a *unique* evaluation context and a *unique* subexpression under a particular reduction strategy. For the same reason, not every expression with a hole in it is a valid evaluation context. For example, $(e_1\ e_2)\ \Box$ 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:

$$\text{evaluation context} \qquad \kappa \quad ::= \quad \Box \mid \kappa\ e \mid (\lambda x\!:\!A.\, e)\ \kappa \mid \mathsf{if}\ \kappa\ \mathsf{then}\ e\ \mathsf{else}\ e$$

$\kappa\ e$ 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 $\mathsf{if}\ \kappa\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ is an evaluation context for $\mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ 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:

$$\begin{array}{rcl}
\Box[\![(\lambda x\!:\!A.\, e'')\ v]\!] & = & (\lambda x\!:\!A.\, e'')\ v \\
(\mathsf{if}\ \Box\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)[\![(\lambda x\!:\!A.\, e'')\ v]\!] & = & (\mathsf{if}\ (\lambda x\!:\!A.\, e'')\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \\
((\mathsf{if}\ \Box\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ e')[\![(\lambda x\!:\!A.\, e'')\ v]\!] & = & (\mathsf{if}\ (\lambda x\!:\!A.\, e'')\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ e'
\end{array}$$

A formal definition of $\kappa[\![e]\!]$ is given as follows:

$$\begin{array}{rcl}
\Box[\![e]\!] & = & e \\
(\kappa\ e')[\![e]\!] & = & \kappa[\![e]\!]\ e' \\
((\lambda x\!:\!A.\, e')\ \kappa)[\![e]\!] & = & (\lambda x\!:\!A.\, e')\ \kappa[\![e]\!] \\
(\mathsf{if}\ \kappa\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)[\![e]\!] & = & \mathsf{if}\ \kappa[\![e]\!]\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2
\end{array}$$

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 subexpression $e$. Since $\kappa[\![e]\!]$ is known to reduce to another expression, $e$ must also reduce to another expression $e'$. We write $e \mapsto_\beta e'$ to indicate that the reduction of $e$ to $e'$ uses the rule $App$, $If_{\mathsf{true}}$, or $If_{\mathsf{false}}$. Then the following reduction rule alone is enough to completely

$$\begin{array}{lrcl} \text{evaluation context} & \kappa & ::= & \Box \mid \kappa\ e \mid (\lambda x\!:\!A.\ e)\ \kappa \mid \mathsf{if}\ \kappa\ \mathsf{then}\ e\ \mathsf{else}\ e \end{array}$$

$$\begin{array}{rcl} (\lambda x\!:\!A.\ e)\ v & \mapsto_\beta & [v/x]e \\ \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \mapsto_\beta & e_1 \\ \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \mapsto_\beta & e_2 \end{array}$$

$$\frac{e \mapsto_\beta e'}{\kappa[\![e]\!] \mapsto \kappa[\![e']\!]}\ Red_\beta$$

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

$$\begin{array}{lrcl} \text{evaluation context} & \kappa & ::= & \Box \mid \kappa\ e \mid \mathsf{if}\ \kappa\ \mathsf{then}\ e\ \mathsf{else}\ e \end{array}$$

$$\begin{array}{rcl} (\lambda x\!:\!A.\ e)\ e' & \mapsto_\beta & [e'/x]e \\ \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \mapsto_\beta & e_1 \\ \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \mapsto_\beta & e_2 \end{array}$$

$$\frac{e \mapsto_\beta e'}{\kappa[\![e]\!] \mapsto \kappa[\![e']\!]}\ Red_\beta$$

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

specify a reduction strategy because the order of reduction is implicitly determined by the definition of evaluation contexts:

$$\frac{e \mapsto_\beta e'}{\kappa[\![e]\!] \mapsto \kappa[\![e']\!]}\ Red_\beta$$

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

$$\begin{array}{rcl} (\lambda x\!:\!A.\ e)\ v & \mapsto_\beta & [v/x]e \\ \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \mapsto_\beta & e_1 \\ \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \mapsto_\beta & e_2 \end{array}$$

Figure 1.1 shows how to use evaluation contexts to specify the call-by-value operational semantics for the simply typed $\lambda$-calculus. 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 $\mapsto_\beta$, as shown in Figure 1.2.

## 1.2 Extensions to the simply typed $\lambda$-calculus

With a proper understanding of evaluation contexts, it should be straightforward to incorporate those reduction rules in Chapter **??** into the definition of evaluation contexts and the reduction relation $\mapsto_\beta$. The reader is encouraged to try to augment the definition of evaluation contexts and the reduction relation $\mapsto_\beta$. See Figures 1.3 and 1.4 for the result.

## 1.3 Type safety

As usual, type safety consists of progress and type preservation:

**Theorem 1.1 (Progress).** *If $\cdot \vdash e : A$ for some type $A$, then either $e$ is a value or there exist $e'$ such that $e \mapsto e'$.*

**Theorem 1.2 (Type preservation).** *If $\Gamma \vdash e : A$ and $e \mapsto e'$, then $\Gamma \vdash e' : A$.*

$$\begin{array}{lrcl}
\text{evaluation context} & \kappa & ::= & \cdots \mid (\kappa, e) \mid (v, \kappa) \mid \mathsf{fst}\ \kappa \mid \mathsf{snd}\ \kappa \mid \\
& & & \mathsf{inl}_A\ \kappa \mid \mathsf{inr}_A\ \kappa \mid \mathsf{case}\ \kappa\ \mathsf{of\ inl}\ x.\, e \mid \mathsf{inr}\ x.\, e
\end{array}$$

$$\begin{array}{rcl}
\mathsf{fst}\ (v_1, v_2) & \mapsto_\beta & v_1 \\
\mathsf{snd}\ (v_1, v_2) & \mapsto_\beta & v_2 \\
\mathsf{case\ inl}_A\ v\ \mathsf{of\ inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 & \mapsto_\beta & [v/x_1]e_1 \\
\mathsf{case\ inr}_A\ v\ \mathsf{of\ inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 & \mapsto_\beta & [v/x_2]e_2 \\
\mathsf{fix}\ x\!:\!A.\, e & \mapsto_\beta & [\mathsf{fix}\ x\!:\!A.\, e/x]e
\end{array}$$

Figure 1.3: Extension for the eager reduction strategy

$$\begin{array}{lrcl}
\text{evaluation context} & \kappa & ::= & \cdots \mid \mathsf{fst}\ \kappa \mid \mathsf{snd}\ \kappa \mid \mathsf{case}\ \kappa\ \mathsf{of\ inl}\ x.\, e \mid \mathsf{inr}\ x.\, e
\end{array}$$

$$\begin{array}{rcl}
\mathsf{fst}\ (e_1, e_2) & \mapsto_\beta & e_1 \\
\mathsf{snd}\ (e_1, e_2) & \mapsto_\beta & e_2 \\
\mathsf{case\ inl}_A\ e\ \mathsf{of\ inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 & \mapsto_\beta & [e/x_1]e_1 \\
\mathsf{case\ inr}_A\ e\ \mathsf{of\ inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 & \mapsto_\beta & [e/x_2]e_2 \\
\mathsf{fix}\ x\!:\!A.\, e & \mapsto_\beta & [\mathsf{fix}\ x\!:\!A.\, e/x]e
\end{array}$$

Figure 1.4: Extension for the lazy reduction strategy

Since the rule $Red_\beta$ uses not only a subexpression of a given expression but also an evaluation context for it, the proof of type safety requires a new typing judgments for evaluation contexts. We write $\Gamma \vdash \kappa : A \Rightarrow C$ to mean that given an expression of type $A$, the evaluation context $\kappa$ produces an expression of type $C$:

$$\Gamma \vdash \kappa : A \Rightarrow C \qquad \Leftrightarrow \qquad \textit{if } \Gamma \vdash e : A, \textit{ then } \Gamma \vdash \kappa[\![e]\!] : C$$

We write $\kappa : A \Rightarrow C$ for $\cdot \vdash \kappa : A \Rightarrow C$.

The following inference rules are all admissible under the above definition of $\Gamma \vdash \kappa : A \Rightarrow C$. That is, we can prove that the premises imply the conclusion in each inference rule.

$$\frac{}{\Gamma \vdash \Box : A \Rightarrow A}\ \Box_{\mathsf{ctx}} \qquad \frac{\Gamma \vdash \kappa : A \Rightarrow B \to C \quad \Gamma \vdash e : B}{\Gamma \vdash \kappa\ e : A \Rightarrow C}\ \mathsf{Lam}_{\mathsf{ctx}}$$

$$\frac{\Gamma \vdash \lambda x\!:\!B.\, e : B \to C \quad \Gamma \vdash \kappa : A \Rightarrow B}{\Gamma \vdash (\lambda x\!:\!B.\, e)\ \kappa : A \Rightarrow C}\ \mathsf{Arg}_{\mathsf{ctx}}$$

$$\frac{\Gamma \vdash \kappa : A \Rightarrow \mathsf{bool} \quad \Gamma \vdash e_1 : C \quad \Gamma \vdash e_2 : C}{\Gamma \vdash \mathsf{if}\ \kappa\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A \Rightarrow C}\ \mathsf{If}_{\mathsf{ctx}}$$

**Proposition 1.3.** *The rules* $\Box_{\mathsf{ctx}}$, $\mathsf{Lam}_{\mathsf{ctx}}$, $\mathsf{Arg}_{\mathsf{ctx}}$, *and* $\mathsf{If}_{\mathsf{ctx}}$ *are admissible.*

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

Case $\dfrac{\Gamma \vdash \kappa : A \Rightarrow B \to C \quad \Gamma \vdash e : B}{\Gamma \vdash \kappa\ e : A \Rightarrow C}\ \mathsf{Lam}_{\mathsf{ctx}}$

| | |
|---|---|
| $\Gamma \vdash \kappa : A \Rightarrow B \to C$ and $\Gamma \vdash e : B$ | assumptions |
| $\Gamma \vdash e' : A$ | assumption |
| $\Gamma \vdash \kappa[\![e']\!] : B \to C$ | from $\Gamma \vdash \kappa : A \Rightarrow B \to C$ and $\Gamma \vdash e' : A$ |
| $\Gamma \vdash \kappa[\![e']\!]\ e : C$ | by the rule $\to$E |
| $\Gamma \vdash (\kappa\ e)[\![e']\!] : C$ | from $\kappa[\![e']\!]\ e = (\kappa\ e)[\![e']\!]$ |
| $\Gamma \vdash \kappa\ e : A \Rightarrow C$ | from $\Gamma \vdash e' : A$ and $\Gamma \vdash (\kappa\ e)[\![e']\!] : C$ |

□

The proof of Theorem 1.1 is similar to the proof of Theorem **??**. The proof of Theorem 1.2 uses the following lemma whose proof uses Lemma **??**:

**Lemma 1.4.** *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 $\kappa = \kappa'\ e'$ | |
| $\Gamma \vdash \kappa[\![e]\!] : C$ | assumption |
| $\Gamma \vdash (\kappa'[\![e]\!])\ e' : C$ | $\kappa[\![e]\!] = (\kappa'[\![e]\!])\ e'$ |
| $\Gamma \vdash \kappa'[\![e]\!] : B \to C$ and $\Gamma \vdash e' : B$ for some type $B$ | by Lemma **??** |
| $\Gamma \vdash e : A$ and $\Gamma \vdash \kappa' : A \Rightarrow B \to C$ for some type $A$ | by IH on $\kappa'$ |
| $\Gamma \vdash \kappa'\ e' : A \Rightarrow C$ | by the rule $\mathsf{Lam}_{\mathsf{ctx}}$ |
| $\Gamma \vdash \kappa : A \Rightarrow C$ | □ |

**Exercise 1.5.** Prove Theorems 1.1 and 1.2.

## 1.4 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 $Red_\beta$ tacitly assumes an automatic decomposition of a given expression into a unique evaluation context $\kappa$ and a unique subexpression $e$, which may in fact require an explicit analysis of the given expression in several steps. For example, in order to rewrite

$$e = (\mathsf{if}\ (\lambda x{:}A.\, e'')\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ e'$$

as $((\mathsf{if}\ \Box\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ e')[\![(\lambda x{:}A.\, e'')\ v]\!]$, we would analyze $e$ as follows:

$$\begin{array}{rcl} e & = & \Box[\![(\mathsf{if}\ (\lambda x{:}A.\, e'')\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ e']\!] \\ & = & (\Box\ e')[\![\mathsf{if}\ (\lambda x{:}A.\, e'')\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] \\ & = & ((\mathsf{if}\ \Box\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ e')[\![(\lambda x{:}A.\, e'')\ v]\!] \end{array}$$

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*, where each frame corresponds to a specific step in an analysis of a given expression:

$$\begin{array}{lrcl} \mathsf{frame} & \phi & ::= & \Box\ e \mid (\lambda x{:}A.\, e)\ \Box \mid \mathsf{if}\ \Box\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\ \mathsf{stack} & \sigma & ::= & \Box \mid \sigma; \phi \end{array}$$

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:

$$\begin{array}{rcl} \Box[\![e]\!] & = & e \\ (\sigma; \phi)[\![e]\!] & = & \sigma[\![\phi[\![e]\!]]\!] \end{array}$$

If we write $\sigma$ as $\Box; \phi_1; \phi_2; \cdots; \phi_n$ for $n \geq 0$, $\sigma[\![e]\!]$ may be written as

$$\Box[\![\phi_1[\![\phi_2[\![\cdots[\![\phi_n[\![e]\!]]\!]\cdots]\!]]\!]]\!].$$

Note that the top frame of a stack $\sigma; \phi$ is $\phi$; the bottom of a stack is always $\Box$.

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 of the abstract machine C:

$$\mathsf{state} \qquad s \quad ::= \quad \sigma \blacktriangleright e \mid \sigma \blacktriangleleft v$$

- $\sigma \blacktriangleright e$ means that the machine is currently reducing $\sigma[\![e]\!]$, but has yet to analyze $e$.
- $\sigma \blacktriangleleft v$ means that the machine is currently reducing $\sigma[\![v]\!]$ and has already analyzed $v$.

If an expression $e$ evaluates to a value $v$, the initial state of the machine would be $\Box \blacktriangleright e$ and the final state $\Box \blacktriangleleft v$.

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

$$\frac{}{\sigma \blacktriangleright v \mapsto_{\mathsf{C}} \sigma \blacktriangleleft v}\ \mathit{Val}_{\mathsf{C}}$$

$$\frac{}{\sigma \blacktriangleright e_1\ e_2 \mapsto_{\mathsf{C}} \sigma; \Box\ e_2 \blacktriangleright e_1}\ \mathit{Lam}_{\mathsf{C}} \qquad \frac{}{\sigma; \Box\ e_2 \blacktriangleleft \lambda x{:}A.\, e \mapsto_{\mathsf{C}} \sigma; (\lambda x{:}A.\, e)\ \Box \blacktriangleright e_2}\ \mathit{Arg}_{\mathsf{C}}$$

$$\frac{}{\sigma; (\lambda x{:}A.\, e)\ \Box \blacktriangleleft v \mapsto_{\mathsf{C}} \sigma \blacktriangleright [v/x]e}\ \mathit{App}_{\mathsf{C}}$$

$$\frac{}{\sigma \blacktriangleright \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto_{\mathsf{C}} \sigma; \mathsf{if}\ \Box\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \blacktriangleright e}\ \mathit{If}_{\mathsf{C}}$$

$$\frac{}{\sigma; \mathsf{if}\ \Box\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \blacktriangleleft \mathsf{true} \mapsto_{\mathsf{C}} \sigma \blacktriangleright e_1}\ \mathit{If}_{\mathsf{trueC}}$$

$$\frac{}{\sigma; \mathsf{if}\ \Box\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \blacktriangleleft \mathsf{false} \mapsto_{\mathsf{C}} \sigma \blacktriangleright e_2}\ \mathit{If}_{\mathsf{falseC}}$$

An example of a reduction sequence is shown below; note that it begins with a state $\Box \blacktriangleright e$ and ends with a state $\Box \blacktriangleleft v$:

$$\begin{array}{lll}
 & \Box \blacktriangleright (\mathsf{if}\ (\lambda x{:}\mathsf{bool}.\, x)\ \mathsf{true}\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z)\ \mathsf{true} & \mathit{Lam}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box; \Box\ \mathsf{true} \blacktriangleright \mathsf{if}\ (\lambda x{:}\mathsf{bool}.\, x)\ \mathsf{true}\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z & \mathit{If}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box; \Box\ \mathsf{true}; \mathsf{if}\ \Box\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z \blacktriangleright (\lambda x{:}\mathsf{bool}.\, x)\ \mathsf{true} & \mathit{Lam}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box; \Box\ \mathsf{true}; \mathsf{if}\ \Box\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z; \Box\ \mathsf{true} \blacktriangleright \lambda x{:}\mathsf{bool}.\, x & \mathit{Val}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box; \Box\ \mathsf{true}; \mathsf{if}\ \Box\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z; \Box\ \mathsf{true} \blacktriangleleft \lambda x{:}\mathsf{bool}.\, x & \mathit{Arg}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box; \Box\ \mathsf{true}; \mathsf{if}\ \Box\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z; (\lambda x{:}\mathsf{bool}.\, x)\ \Box \blacktriangleright \mathsf{true} & \mathit{Val}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box; \Box\ \mathsf{true}; \mathsf{if}\ \Box\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z; (\lambda x{:}\mathsf{bool}.\, x)\ \Box \blacktriangleleft \mathsf{true} & \mathit{App}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box; \Box\ \mathsf{true}; \mathsf{if}\ \Box\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z \blacktriangleright \mathsf{true} & \mathit{Val}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box; \Box\ \mathsf{true}; \mathsf{if}\ \Box\ \mathsf{then}\ \lambda y{:}\mathsf{bool}.\, y\ \mathsf{else}\ \lambda z{:}\mathsf{bool}.\, z \blacktriangleleft \mathsf{true} & \mathit{If}_{\mathsf{trueC}} \\
\mapsto_{\mathsf{C}} & \Box; \Box\ \mathsf{true} \blacktriangleright \lambda y{:}\mathsf{bool}.\, y & \mathit{Val}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box; \Box\ \mathsf{true} \blacktriangleleft \lambda y{:}\mathsf{bool}.\, y & \mathit{Arg}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box; (\lambda y{:}\mathsf{bool}.\, y)\ \Box \blacktriangleright \mathsf{true} & \mathit{Val}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box; (\lambda y{:}\mathsf{bool}.\, y)\ \Box \blacktriangleleft \mathsf{true} & \mathit{App}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box \blacktriangleright \mathsf{true} & \mathit{Val}_{\mathsf{C}} \\
\mapsto_{\mathsf{C}} & \Box \blacktriangleleft \mathsf{true} &
\end{array}$$

## 1.5 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.** *$e \mapsto^* v$ if and only if $\Box \blacktriangleright e \mapsto_{\mathsf{C}}^* \Box \blacktriangleleft v$.*

A more general version of the theorem allows any stack $\sigma$ in place of $\Box$, but we do not prove it here. For the sake of simplicity, we also do not consider expressions of type 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:

$$\begin{array}{rcl} \Box[\![\kappa']\!] & = & \kappa' \\ (\kappa\ e)[\![\kappa']\!] & = & \kappa[\![\kappa']\!]\ e \\ ((\lambda x{:}A.\, e)\ \kappa)[\![\kappa']\!] & = & (\lambda x{:}A.\, e)\ \kappa[\![\kappa']\!] \end{array}$$

$$\begin{array}{rcl} \Box[\![\kappa]\!] & = & \kappa \\ (\sigma;\phi)[\![\kappa]\!] & = & \sigma[\![\phi[\![\kappa]\!]]\!] \end{array}$$

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 $\kappa = \kappa''\ e''$:
$(\kappa''\ e'')[\![\kappa'[\![e]\!]]\!] = \kappa''[\![\kappa'[\![e]\!]]\!]\ e'' = \kappa''[\![\kappa']\!][\![e]\!]\ e'' = (\kappa''[\![\kappa']\!]\ e'')[\![e]\!] = (\kappa''\ e'')[\![\kappa']\!][\![e]\!]$ □

**Proposition 1.8.** $\sigma[\![\kappa[\![e]\!]]\!] = \sigma[\![\kappa]\!][\![e]\!]$.

*Proof.* By structural induction on $\sigma$. The second case uses Proposition 1.7.

Case $\sigma = \Box$:
$\Box[\![\kappa[\![e]\!]]\!] = \kappa[\![e]\!] = \Box[\![\kappa]\!][\![e]\!]$.

Case $\sigma = \sigma'; \phi$:
$(\sigma';\phi)[\![\kappa[\![e]\!]]\!] = \sigma'[\![\phi[\![\kappa[\![e]\!]]\!]]\!] = \sigma'[\![\phi[\![\kappa]\!][\![e]\!]]\!] = \sigma'[\![\phi[\![\kappa]\!]]\!][\![e]\!] = (\sigma';\phi)[\![\kappa]\!][\![e]\!]$. □

**Lemma 1.9.** *For $\sigma$ and $\kappa$, there exists $\sigma'$ such that $\sigma \blacktriangleright \kappa[\![e]\!] \mapsto_{\mathsf{C}}^{*} \sigma' \blacktriangleright e$ 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 $\kappa = \kappa'\ e'$:
$\sigma \blacktriangleright (\kappa'\ e')[\![e]\!] = \sigma \blacktriangleright \kappa'[\![e]\!]\ e' \mapsto_{\mathsf{C}} \sigma; \Box\ e' \blacktriangleright \kappa'[\![e]\!]$ by the rule $Lam_{\mathsf{C}}$
$\sigma; \Box\ e' \blacktriangleright \kappa'[\![e]\!] \mapsto_{\mathsf{C}}^{*} \sigma' \blacktriangleright e$ and $(\sigma; \Box\ e')[\![\kappa']\!] = \sigma'[\![\Box]\!]$ by IH
$\sigma[\![\kappa]\!] = \sigma[\![\kappa'\ e']\!] = \sigma[\![(\Box\ e')[\![\kappa']\!]]\!] = (\sigma; \Box\ e')[\![\kappa']\!] = \sigma'[\![\Box]\!]$ □

**Lemma 1.10.** *Suppose $\sigma[\![e]\!] = \kappa[\![f\ v]\!]$ where $f$ is a $\lambda$-abstraction. Then one of the following cases holds:*
*(1) $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^{*} \sigma' \blacktriangleright \kappa'[\![f\ v]\!]$ 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
$\sigma \blacktriangleright e = \Box \blacktriangleright \kappa[\![f\ v]\!]$
(1) $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^{*} \Box \blacktriangleright \kappa[\![f\ v]\!]$ 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) $\sigma' \blacktriangleright e\ e' \mapsto_{\mathsf{C}}^* \sigma'' \blacktriangleright \kappa'[\![f\ v]\!]$ and $\sigma''[\![\kappa']\!] = \kappa$: by IH on $\sigma'$

$\sigma' \blacktriangleright e\ e' \mapsto_{\mathsf{C}} \sigma'; \Box\ e' \blacktriangleright e \mapsto_{\mathsf{C}}^* \sigma'' \blacktriangleright \kappa'[\![f\ v]\!]$ assumption
(1) $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^* \sigma'' \blacktriangleright \kappa'[\![f\ v]\!]$ and $\sigma''[\![\kappa']\!] = \kappa$

$\sigma' = \sigma''$ and $e\ e' = \kappa'[\![f\ v]\!]$, and $\kappa' = \Box$ assumption
$e = f$ and $e' = v$
(2) $\sigma = \sigma'; \Box\ v$ and $e = f$ and $\sigma'[\![\Box]\!] = \sigma''[\![\kappa']\!] = \kappa$

$\sigma' = \sigma''$ and $e\ e' = \kappa'[\![f\ v]\!]$, and $\kappa' = \kappa''\ e'$ assumption
$e = \kappa''[\![f\ v]\!]$
(1) $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^* \sigma \blacktriangleright \kappa''[\![f\ v]\!]$ and
$\sigma[\![\kappa'']\!] = (\sigma''; \Box\ e')[\![\kappa'']\!] = \sigma''[\![\kappa''\ e']\!] = \sigma''[\![\kappa']\!] = \kappa$

$\sigma' = \sigma''$ and $e\ e' = \kappa'[\![f\ v]\!]$, and $\kappa' = e\ \kappa''$ assumption
$e$ is a $\lambda$-abstraction and $e' = \kappa''[\![f\ v]\!]$
(1) $\sigma \blacktriangleright e = \sigma'; \Box\ e' \blacktriangleright e \mapsto_{\mathsf{C}} \sigma'; \Box\ e' \blacktriangleleft e \mapsto_{\mathsf{C}} \sigma'; e\ \Box \blacktriangleright e' = \sigma'; e\ \Box \blacktriangleright \kappa''[\![f\ v]\!]$
and $(\sigma'; e\ \Box)[\![\kappa'']\!] = \sigma'[\![e\ \kappa'']\!] = \sigma''[\![\kappa']\!] = \kappa$

Subcases (2) and (3) impossible $e\ e' \neq f$ and $e\ e' \neq v$ □

**Lemma 1.11.** *Suppose $\sigma[\![e]\!] = \kappa[\![f\ v]\!]$ where $f$ is a $\lambda$-abstraction and $f\ v \mapsto_\beta e'$. Then $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^* \sigma^* \blacktriangleright e'$ and $\sigma^*[\![e']\!] = \kappa[\![e']\!]$.*

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

(1) $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^* \sigma' \blacktriangleright \kappa'[\![f\ v]\!]$ and $\sigma'[\![\kappa']\!] = \kappa$

$$\begin{array}{lll}
\sigma \blacktriangleright e & \mapsto_{\mathsf{C}}^* & \sigma' \blacktriangleright \kappa'[\![f\ v]\!] \\
& \mapsto_{\mathsf{C}}^* & \sigma'' \blacktriangleright f\ v \qquad \text{where } \sigma'[\![\kappa']\!] = \sigma''[\![\Box]\!] \text{ by Lemma 1.9} \\
& \mapsto_{\mathsf{C}} & \sigma''; \Box\ v \blacktriangleright f \\
& \mapsto_{\mathsf{C}} & \sigma''; \Box\ v \blacktriangleleft f \\
& \mapsto_{\mathsf{C}} & \sigma''; f\ \Box \blacktriangleright v \\
& \mapsto_{\mathsf{C}} & \sigma''; f\ \Box \blacktriangleleft v \\
& \mapsto_{\mathsf{C}} & \sigma'' \blacktriangleright e'
\end{array}$$

$\sigma''[\![e']\!] = \sigma''[\![\Box[\![e']\!]]\!] = \sigma''[\![\Box]\!][\![e']\!] = \sigma'[\![\kappa']\!][\![e']\!] = \kappa[\![e']\!]$ by Proposition 1.8
We let $\sigma^* = \sigma''$.

(2) $\sigma = \sigma'; \Box\ v$ and $e = f$ and $\sigma'[\![\Box]\!] = \kappa$

$$\begin{array}{lll}
\sigma \blacktriangleright e & = & \sigma'; \Box\ v \blacktriangleright f \\
& \mapsto_{\mathsf{C}}^* & \sigma' \blacktriangleright e'
\end{array}$$

$\sigma'[\![e']\!] = \sigma'[\![\Box[\![e']\!]]\!] = \sigma'[\![\Box]\!][\![e']\!] = \kappa[\![e']\!]$ by Proposition 1.8
We let $\sigma^* = \sigma'$.

(3) $\sigma = \sigma'; f\ \Box$ and $e = v$ and $\sigma'[\![\Box]\!] = \kappa$

$$\begin{array}{lll}
\sigma \blacktriangleright e & = & \sigma'; f\ \Box \blacktriangleright v \\
& \mapsto_{\mathsf{C}}^* & \sigma' \blacktriangleright e'
\end{array}$$

$\sigma'[\![e']\!] = \sigma'[\![\Box[\![e']\!]]\!] = \sigma'[\![\Box]\!][\![e']\!] = \kappa[\![e']\!]$ by Proposition 1.8
We let $\sigma^* = \sigma'$. □

**Corollary 1.12.** *Suppose $e_1 \mapsto e_2$ and $\sigma[\![e]\!] = e_1$. Then there exist $\sigma'$ and $e'$ such that $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^* \sigma' \blacktriangleright e'$ and $\sigma'[\![e']\!] = e_2$.*

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

**Proposition 1.13.** *Suppose* $e \mapsto^* v$ *and* $\sigma[\![e']\!] = e$. *Then* $\sigma \blacktriangleright e' \mapsto_{\mathsf{C}}^* \square \blacktriangleleft v$.

**Corollary 1.14.** *If* $e \mapsto^* v$, *then* $\square \blacktriangleright e \mapsto_{\mathsf{C}}^* \square \blacktriangleleft v$.

**Proposition 1.15.**
*If* $\sigma \blacktriangleright e \mapsto_{\mathsf{C}} \sigma' \blacktriangleright e'$, *then* $\sigma[\![e]\!] \mapsto^* \sigma'[\![e']\!]$.
*If* $\sigma \blacktriangleright e \mapsto_{\mathsf{C}} \sigma' \blacktriangleleft v'$, *then* $\sigma[\![e]\!] \mapsto^* \sigma'[\![v']\!]$.

**Corollary 1.16.**
*If* $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^* \sigma' \blacktriangleright e'$, *then* $\sigma[\![e]\!] \mapsto^* \sigma'[\![e']\!]$.
*If* $\sigma \blacktriangleright e \mapsto_{\mathsf{C}}^* \sigma' \blacktriangleleft v'$, *then* $\sigma[\![e]\!] \mapsto^* \sigma'[\![v']\!]$.

**Corollary 1.17.** *If* $\square \blacktriangleright e \mapsto_{\mathsf{C}}^* \square \blacktriangleleft v$, *then* $e \mapsto^* v$.

Corollaries 1.14 and 1.17 prove Theorem 1.6.

## 1.6 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.

$$\frac{\sigma[\![\square]\!] : A \Rightarrow C \quad \cdot \vdash e : A}{\sigma \blacktriangleright e \ \text{okay}}\ \mathsf{Okay}_{\blacktriangleright} \qquad \frac{\sigma[\![\square]\!] : A \Rightarrow C \quad \cdot \vdash v : A}{\sigma \blacktriangleleft v \ \text{okay}}\ \mathsf{Okay}_{\blacktriangleleft}$$

$$\frac{\cdot \vdash v : A}{\square \blacktriangleleft v \ \text{stop}}\ \mathsf{Stop}_{\blacktriangleleft}$$

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* $s \mapsto_{\mathsf{C}} s'$.
*If* $s$ okay *and* $s \mapsto_{\mathsf{C}} s'$, *then* $s'$ okay.

## 1.7 Exercises

**Exercise 1.19.** Extend the abstract machine C for product types, sum types, and the fixed point construct.

**Exercise 1.20.** Prove Theorem 1.18.