Chapter 1

Simply typed λ-calculus

This chapter presents the *simply typed λ-calculus*, an extension of the λ-calculus with *types*. Since the λ-calculus in the previous chapter does not use types, we refer to it as the *untyped* λ-calculus so that we can differentiate it from the simply typed λ-calculus.

Unlike the untyped λ-calculus in which base types (such as boolean and integers) are simulated with λ-abstractions, the simply typed λ-calculus assumes a fixed set of base types with primitive constructs. For example, we may choose to include a base type bool with boolean constants true and false and a conditional construct $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. Thus the simply typed λ-calculus may be thought of as not just a core calculus for investigating the expressive power but indeed a subset of a functional language. Then any expression in the simply typed λ-calculus can be literally translated in a functional language such as SML.

As with the untyped λ-calculus, we first formulate the abstract syntax and operational semantics of the simply typed λ-calculus. The difference in the operational semantics is nominal because types play no role in reducing expressions. A major change arises from the introduction of a *type system*, a collection of judgments and inference rules for assigning types to expressions. The type assigned to an expression determines the form of value to which the expression evaluates. For example, an expression of type bool may evaluate to either true or false, but nothing else.

The focus of the present chapter is on *type safety*, the most basic property of a type system that an expression with a valid type, or a *well-typed* expression, cannot go wrong at runtime. Since an expression is assigned a type at compile time and type safety ensures that a well-typed expression is well-behaved at runtime, we do not need the trial and error method (of running a program to locate the source of a bug in it) in order to detect fatal bugs such as adding memory addresses, subtracting an integer from a string, using an integer as a destination address in a function invocation, and so forth. Since it is often these simple (and stupid) bugs that cause considerable delay in software development, type safety offers a huge advantage over those programming languages without type systems or with type systems that fail to support type safety. Type safety is also the reason behind the phenomenon that programs that successfully compile run correctly in most cases.

Every extension to the simply typed λ-calculus discussed in this course will preserve type safety. We definitely do not want to squander our time developing a programming language as uncivilized as C!

1.1 Abstract syntax

The abstract syntax for the simply typed λ-calculus is given as follows:

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

A type is either a *base type* P or a *function type* $A \rightarrow A'$. A base type is a type whose primitive constructs are given as part of the definition. Here we use a boolean type bool as a base type with which three primitive constructs are associated: boolean constants true and false and a conditional construct $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. A function type $A \rightarrow A'$ describes those functions taking arguments of type A and returning results of type A'. We use metavariables A, B, C for types.

It is important that the simply typed λ-calculus does not stipulate specific base types. In other words, the simply typed λ-calculus is just a framework for functional languages whose type system is extensible with additional base types. For example, the definition above considers bool as the only base type, but it should also be clear how to extend the definition with another base type (*e.g.*, an integer type int with integer constants and arithmetic operators). On the other hand, the simply typed λ-calculus must have at least one base type. Otherwise the set P of base types is empty, which in turn makes the set A of types empty. Then we would never be able to create an expression with a valid type!

As in the untyped λ-calculus, expressions include variables, λ-abstractions or functions, and applications. A λ-abstraction $\lambda x\!:\!A.\,e$ now explicitly specifies the type A of its formal argument x. If $\lambda x\!:\!A.\,e$ is applied to an expression of a different type A' (*i.e.*, $A \neq A'$), the application does not typecheck and thus has no type, as will be seen in Section 1.3. We say that variable x is bound to type A in a λ-abstraction $\lambda x\!:\!A.\,e$, or that a λ-abstraction $\lambda x\!:\!A.\,e$ binds variable x to type A.

1.2 Operational semantics

The development of the operational semantics of the simply typed λ-calculus is analogous to the case for the untyped λ-calculus: we define a mapping $FV(e)$ to calculate the set of free variables in e, a capture-avoiding substitution $[e'/x]e$, and a reduction judgment $e \mapsto e'$ with reduction rules. Since the simply typed λ-calculus is no different from the untyped λ-calculus except for its use of a type system, its operational semantics reverts to the operational semantics of the untyped λ-calculus if we ignore types in expressions.

A mapping $FV(e)$ is defined as follows:

$$
\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(\lambda x\!:\!A.\,e) & = & FV(e) - \{x\} \\
FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) \\
FV(\mathsf{true}) & = & \varnothing \\
FV(\mathsf{false}) & = & \varnothing \\
FV(\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & FV(e) \cup FV(e_1) \cup FV(e_2)
\end{array}
$$

As in the untyped λ-calculus, we say that an expression is closed if it contains no free variables.

A capture-avoiding substitution $[e'/x]e$ is defined as follows:

$$
\begin{array}{rcll}
[e'/x]x & = & e' & \\
{[e'/x]y} & = & y & \textit{if } x \neq y \\
{[e'/x]\lambda x\!:\!A.\,e} & = & \lambda x\!:\!A.\,e & \\
{[e'/x]\lambda y\!:\!A.\,e} & = & \lambda y\!:\!A.\,[e'/x]e & \textit{if } x \neq y, y \notin FV(e') \\
{[e'/x](e_1\ e_2)} & = & [e'/x]e_1\ [e'/x]e_2 & \\
{[e'/x]\mathsf{true}} & = & \mathsf{true} & \\
{[e'/x]\mathsf{false}} & = & \mathsf{false} & \\
{[e'/x]\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2} & = & \mathsf{if}\ [e'/x]e\ \mathsf{then}\ [e'/x]e_1\ \mathsf{else}\ [e'/x]e_2 &
\end{array}
$$

When a variable capture occurs in $[e'/x]\lambda y\!:\!A.\,e$, we rename the bound variable y using the α-equivalence relation $\equiv_\alpha$. We omit the definition of $\equiv_\alpha$ because it requires no further consideration than the definition given in Chapter **??**.

As with the untyped λ-calculus, different reduction strategies yield different reduction rules for the reduction judgment $e \mapsto e'$. We choose the call-by-value strategy which lends itself well to extending the simply typed λ-calculus with *computational effects* such as mutable references, exceptions, and continuations (to be discussed in subsequent chapters). Thus we use the following reduction rules:

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

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

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

The rules Lam, Arg, and App are exactly the same as in the untyped λ-calculus except that we use a λ-abstraction of the form $\lambda x\!:\!A.\,e$. (To implement the call-by-name strategy, we remove the rule Arg and rewrite the rule App as $\frac{}{(\lambda x\!:\!A.\,e)\ e' \mapsto [e'/x]e}\ \mathit{App}$.) The three rules If, $\mathit{If}_{\mathsf{true}}$, and $\mathit{If}_{\mathsf{false}}$ combined together specify how to reduce a conditional construct $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$:

- We reduce e to either true or false.
- If e reduces to true, we choose the then branch and begin to reduce e_1.
- If e reduces to false, we choose the else branch and begin to reduce e_2.

As before, we write $\mapsto^*$ for the reflexive and transitive closure of $\mapsto$. We say that e evaluates to v if $e \mapsto^* v$ holds.

1.3 Type system

The goal of this section is to develop a system of inference rules for assigning types to expressions in the simply typed λ-calculus. We use a judgment called a *typing judgment*, and refer to inference rules deducing a typing judgment as *typing rules*. The resultant system is called the *type system* of the simply typed λ-calculus.

To figure out the right form for the typing judgment, let us consider an identity function $\mathsf{id} = \lambda x\!:\!A.\,x$. Intuitively id has a function type $A \rightarrow A$ because it takes an argument of type A and returns a result of the same type. Then how do we determine, or "infer," the type of id? Since id is a λ-abstraction with an argument of type A, all we need is the type of its body. It is easy to see, however, that its body cannot be considered in isolation: without any assumption on the type of its argument x, we cannot infer the type of its body x!

The example of id suggests that it is inevitable to use assumptions on types of variables in typing judgments. Thus we are led to introduce a *typing context* to denote a collection of assumptions on types of variables:

$$\text{typing context} \qquad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A$$

$\cdot$ denotes an empty typing context, and a *type binding* $x : A$ means that variable x has type A. For the sake of simplicity, we assume that variables in a typing context are all distinct. That is, $\Gamma, x : A$ is not defined if Γ contains another type binding of the form $x : A'$, or simply if $x : A' \in \Gamma$.

The type system uses the following form of typing judgment:[1]

$$\Gamma \vdash e : A \qquad \Leftrightarrow \qquad \textit{expression } e \textit{ has type } A \textit{ under typing context } \Gamma$$

[1] A typing judgment $\Gamma \vdash e : A$ is an example of a *hypothetical judgment* which deduces a "judgment" $e : A$ using each "judgment" $x_i : A_i$ in Γ as a hypothesis. From this point of view, the turnstile symbol $\vdash$ is just a syntactic device which plays no semantic role at all. Although the notion of hypothetical judgment is of great significance in the study of logic, I do not find it particularly useful in helping students to understand the type system of the simply typed λ-calculus.

$\Gamma \vdash e : A$ means that if we use each type binding $x : A$ in Γ as an assumption, we can show that expression e has type A. An easy way to understand the role of Γ is by thinking of it as a collection of type bindings for free variables in e, although Γ may also contain type bindings for those variables not found in e. For example, a closed expression e of type A needs a typing judgment $\cdot \vdash e : A$ with an empty typing context (because it contains no free variables), whereas an expression e' with a free variable x needs a typing judgment $\Gamma \vdash e' : A'$ where Γ contains a type binding $x : B$ for some type B.

With the above interpretation of typing judgments, we can now explain the typing rules for the simply typed λ-calculus:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \mathsf{Var} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x{:}A.\, e : A \to B}\ {\to}\mathsf{I} \qquad \frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B}\ {\to}\mathsf{E}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ \mathsf{True} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \mathsf{False}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A}\ \mathsf{If}$$

- The rule Var means that a type binding in a typing context is an assumption. Alternatively we may rewrite the rule as follows:
$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}\ \mathsf{Var}$$

- The rule $\to$I says that if e has type B under the assumption that x has type A, then $\lambda x{:}A.\, e$ has type $A \to B$. If we read the rule $\to$I from the premise to the conclusion (*i.e.*, top-down), we "introduce" a function type $A \to B$ from the judgment in the premise, which is the reason why it is called the "$\to$ Introduction rule." Note that if Γ already contains a type binding for variable x (*i.e.*, $x : A' \in \Gamma$), we rename x to a fresh variable by an α-conversion. Hence we may assume without loss of generality that variable clashes never occur in the rule $\to$I.

- The rule $\to$E says that if e has type $A \to B$ and e' has type A, then $e\ e'$ has type B. If we read the rule $\to$E from the premise to the conclusion, we "eliminate" a function type $A \to B$ to produce an expression of a smaller type B, which is the reason why it is called the "$\to$ Elimination rule."

- The rules True and False assign base type bool to boolean constants true and false. Note that typing context Γ is not used because there is no free variable in true and false.

- The rule If says that if e has type bool and both e_1 and e_2 have the same type A, then if e then e_1 else e_2 has type A.

A derivation tree for a typing judgment is called a *typing derivation*. Here are a few examples of valid typing derivations. The first example infers the type of an identify function:

$$\cfrac{\cfrac{x : A \in \Gamma, x : A}{\Gamma, x : A \vdash x : A}\ \mathsf{Var}}{\Gamma \vdash \lambda x{:}A.\, x : A \to A}\ {\to}\mathsf{I}$$

In the second example below, we abbreviate $\Gamma, x : \mathsf{bool}, y_1 : A, y_2 : A$ as Γ'. Note also that $\mathsf{bool} \to A \to A \to A$ is equivalent to $\mathsf{bool} \to (A \to (A \to A))$ because $\to$ is right-associative:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{x : \mathsf{bool} \in \Gamma'}{\Gamma' \vdash x : \mathsf{bool}}\ \mathsf{Var} \quad \cfrac{y_1 : A \in \Gamma'}{\Gamma' \vdash y_1 : A}\ \mathsf{Var} \quad \cfrac{y_2 : A \in \Gamma'}{\Gamma' \vdash y_2 : A}\ \mathsf{Var}}{\Gamma, x : \mathsf{bool}, y_1 : A, y_2 : A \vdash \mathsf{if}\ x\ \mathsf{then}\ y_1\ \mathsf{else}\ y_2 : A}\ \mathsf{If}}{\Gamma, x : \mathsf{bool}, y_1 : A \vdash \lambda y_2{:}A.\, \mathsf{if}\ x\ \mathsf{then}\ y_1\ \mathsf{else}\ y_2 : A \to A}\ {\to}\mathsf{I}}{\Gamma, x : \mathsf{bool} \vdash \lambda y_1{:}A.\, \lambda y_2{:}A.\, \mathsf{if}\ x\ \mathsf{then}\ y_1\ \mathsf{else}\ y_2 : A \to A \to A}\ {\to}\mathsf{I}}{\Gamma \vdash \lambda x{:}\mathsf{bool}.\, \lambda y_1{:}A.\, \lambda y_2{:}A.\, \mathsf{if}\ x\ \mathsf{then}\ y_1\ \mathsf{else}\ y_2 : \mathsf{bool} \to A \to A \to A}\ {\to}\mathsf{I}$$

The third example infers the type of a function composing two functions f and g. We abbreviate $\Gamma, f : A \to B, g : B \to C, x : A$ as Γ':

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{g : B \to C \in \Gamma'}{\Gamma' \vdash g : B \to C}\,\mathsf{Var} \quad \cfrac{\cfrac{f : A \to B \in \Gamma'}{\Gamma' \vdash f : A \to B}\,\mathsf{Var} \quad \cfrac{x : A \in \Gamma'}{\Gamma' \vdash x : A}\,\mathsf{Var}}{\Gamma' \vdash f\ x : B}\,{\to}\mathsf{E}}{\Gamma, f : A \to B, g : B \to C, x : A \vdash g\ (f\ x) : C}\,{\to}\mathsf{E}}{\Gamma, f : A \to B, g : B \to C \vdash \lambda x{:}A.\, g\ (f\ x) : A \to C}\,{\to}\mathsf{I}}{\Gamma, f : A \to B \vdash \lambda g{:}B \to C.\, \lambda x{:}A.\, g\ (f\ x) : (B \to C) \to (A \to C)}\,{\to}\mathsf{I}}{\Gamma \vdash \lambda f{:}A \to B.\, \lambda g{:}B \to C.\, \lambda x{:}A.\, g\ (f\ x) : (A \to B) \to (B \to C) \to (A \to C)}\,{\to}\mathsf{I}$$

We close this section by proving two properties of typing judgments: *permutation* and *weakening*. The permutation property reflects the assumption that a typing context Γ is an unordered set, which means that two typing contexts are identified up to permutation. For example, $\Gamma, x : A, y : B$ is identified with $\Gamma, y : B, x : A$, with $x : A, \Gamma, y : B$, with $x : A, y : B, \Gamma$, and so on. The weakening property says that if we can prove that expression e has type A under typing context Γ, we can also prove it under another typing context Γ' augmenting Γ with a new type binding $x : A$ (because we can just ignore the new type binding $x : A$). These properties are called *structural properties* of typing judgments because they deal with the structure of typing judgments rather than their derivations.[2]

Proposition 1.1 (Permutation). *If $\Gamma \vdash e : A$ and Γ' is a permutation of Γ, then $\Gamma' \vdash e : A$.*

Proof. By rule induction on the judgment $\Gamma \vdash e : A$. □

Proposition 1.2 (Weakening). *If $\Gamma \vdash e : C$, then $\Gamma, x : A \vdash e : C$.*

Proof. By rule induction on the judgment $\Gamma \vdash e : C$. □

1.4 Type safety

In order to determine properties of expressions, we have developed two systems for the simply typed λ-calculus: operational semantics and type system. The operational semantics enables us to find out dynamic properties, namely values, associated with expressions. Values are dynamic properties in the sense that they can be determined only at runtime in general. For this reason, an operational semantics is also called a *dynamic semantics*. In contrast, the type system enables us to find out static properties, namely types, of expressions. Types are static properties in the sense that they are determined at compile time and remain "static" at runtime. For this reason, a type system is also called a *static semantics*.

We have developed the type system independently of the operational semantics. Therefore there remains a possibility that it does not respect the operational semantics, whether intentionally or unintentionally. For example, it may assign different types to two expressions e and e' such that $e \mapsto e'$, which is unnatural because we do not anticipate a change in type when an expression reduces to another expression. Or it may assign a valid type to a nonsensical expression, which is also unnatural because we expect every expression of a valid type to be a valid program. Type safety, the most basic property of a type system, connects the type system with the operational semantics by ensuring that it lives in harmony with the operational semantics. It is often rephrased as *"well-typed expressions cannot go wrong."*

Type safety consists of two theorems: *progress* and *type preservation*. The progress theorem states that a (closed) well-typed expression is not stuck: either it is a value or it reduces to another expression:

Theorem 1.3 (Progress). *If $\cdot \vdash e : A$ for some type A, then either e is a value or there exists e' such that $e \mapsto e'$.*

[2]There is no strict rule on whether a property should be called a theorem or a proposition, although a general rule of thumb is that a property relatively easy to prove is called a proposition. A property that is of great significance is usually called a theorem. (For example, it is Fermat's last theorem rather than Fermat's last proposition.) A lemma is a property proven for facilitating proofs of other theorems, propositions, or lemmas.

The type preservation theorem states that when a well-typed expression reduces, the resultant expression is also well-typed and has the same type; type preservation is also called *subject reduction*:

Theorem 1.4 (Type preservation). *If* $\Gamma \vdash e : A$ *and* $e \mapsto e'$, *then* $\Gamma \vdash e' : A$.

Note that the progress theorem assumes an empty typing context (hence a *closed* well-typed expression e) whereas the type preservation theorem does not. It actually makes sense if we consider whether a reduction judgment $e \mapsto e'$ is part of the conclusion or is given as an assumption. In the case of the progress theorem, we are interested in whether e reduces to another expression or not, provided that it is well-typed. Therefore we use an empty typing context to disallow free variables in e which may make its reduction impossible. If we allowed any typing context Γ, the progress theorem would be downright false, as evidenced by a simple counterexample $e = x$ which is not a value and is irreducible. In the case of the type preservation theorem, we begin with an assumption $e \mapsto e'$. Then there is no reason not to allow free variables in e because we already know that it reduces to another expression e'. Thus we use a metavariable Γ (ranging over all typing contexts) instead of an empty typing context.

Combined together, the two theorems guarantee that a (closed) well-typed expression never reduces to a stuck expression: either it is a value or it reduces to another well-typed expressions. Consider a well-typed expression e such that $\cdot \vdash e : A$ for some type A. If e is already a value, there is no need to reduce it (and hence it is not stuck). If not, the progress theorem ensures that there exists an expression e' such that $e \mapsto e'$, which is also a well-typed expression of the same type A by the type preservation theorem.

Below we prove the two theorems using rule induction. It turns out that a direct proof attempt by rule induction fails, and thus we need a couple of lemmas. These lemmas (*canonical forms* and *substitution*) are so prevalent in programming language theory that their names are worth memorizing.

1.4.1 Proof of progress

The proof of Theorem 1.3 is relatively straightforward: the theorem is written in the form *"If J holds, then $P(J)$ holds,"* and we apply rule induction to the judgment J, which is a typing judgment $\cdot \vdash e : A$. So we begin with an assumption $\cdot \vdash e : A$. If e happens to be a value, the $P(J)$ part holds trivially because the judgment *"e is a value"* holds. Thus we make a stronger assumption $\cdot \vdash e : A$ with e not being a value. Then we analyze the structure of the proof of $\cdot \vdash e : A$, which gives three cases to consider:

$$\frac{x : A \in \cdot}{\cdot \vdash x : A}\ \mathsf{Var} \qquad \frac{\cdot \vdash e_1 : A \to B \quad \cdot \vdash e_2 : A}{\cdot \vdash e_1\ e_2 : B}\ {\to}\mathsf{E} \qquad \frac{\cdot \vdash e_b : \mathsf{bool} \quad \cdot \vdash e_1 : A \quad \cdot \vdash e_2 : A}{\cdot \vdash \mathsf{if}\ e_b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A}\ \mathsf{If}$$

The case Var is impossible because $x : A$ cannot be a member of an empty typing context $\cdot$. That is, the premise $x : A \in \cdot$ is never satisfied. So we are left with the two cases ${\to}\mathsf{E}$ and If. Let us analyze the case ${\to}\mathsf{E}$ in depth. By the principle of rule induction, the induction hypothesis on the first premise $\cdot \vdash e_1 : A \to B$ opens two possibilities:

1. e_1 is a value.

2. e_1 is not a value and reduces to another expression e_1', *i.e.*, $e_1 \mapsto e_1'$.

If the second possibility is the case, we have found an expression to which $e_1\ e_2$ reduces, namely $e_1'\ e_2$:

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ Lam$$

Now what if the first possibility is the case? Since e_1 has type $A \to B$, it is likely to be a λ-abstraction, in which case the induction hypothesis on the second premise $\cdot \vdash e_2 : A$ opens another two possibilities and we use either the rule Arg or the rule App to show the progress property. Unfortunately we do not have a formal proof that e_1 is indeed a λ-abstraction; we know only that e_1 has type $A \to B$ under an empty typing context. Our instinct, however, says that e_1 must be a λ-abstraction because it has type $A \to B$. The following lemma formalizes our instinct on the correct, or "canonical," form of a well-typed value:

Lemma 1.5 (Canonical forms).
If v is a value of type bool*, then v is either* true *or* false*.*
If v is a value of type $A \rightarrow B$, then v is a λ-abstraction $\lambda x\!:\!A.\, e$.

Proof. By case analysis of v. (Not every proof uses rule induction!)

Suppose that v is a value of type bool. The only typing rules that assign a boolean type to a given value are True and False. Therefore v is a boolean constant true or false. Note that the rules Var, $\rightarrow$E, and If may assign a boolean type, but never to a value.

Suppose that v is a value of type $A \rightarrow B$. The only typing rule that assigns a function type to a given value is $\rightarrow$I. Therefore v must be a λ-abstraction of the form $\lambda x\!:\!A.\, e$ (which binds variable x to type A). Note that the rules Var, $\rightarrow$E, and If may assign a function type, but not to a value. $\square$

Now we are ready to prove the progress theorem:

Proof of Theorem 1.3. By rule induction on the judgment $\cdot \vdash e : A$.

If e is already a value, we need no further consideration. Therefore we assume that e is not a value. Then there are three cases to consider.

Case $\dfrac{x : A \in \cdot}{\cdot \vdash x : A}\,\mathsf{Var}$ where $e = x$:
impossible — from $x : A \notin \cdot$

Case $\dfrac{\cdot \vdash e_1 : A \rightarrow B \quad \cdot \vdash e_2 : A}{\cdot \vdash e_1\; e_2 : B}\,\rightarrow\!\mathsf{E}$ where $e = e_1\; e_2$:
e_1 is a value or there exists e_1' such that $e_1 \mapsto e_1'$
by induction hypothesis on $\cdot \vdash e_1 : A \rightarrow B$
e_2 is a value or there exists e_2' such that $e_2 \mapsto e_2'$
by induction hypothesis on $\cdot \vdash e_2 : A$

Subcase: e_1 is a value and e_2 is a value
$e_1 = \lambda x\!:\!A.\, e_1'$ — by Lemma 1.5
$e_2 = v_2$ — because e_2 is a value
$e_1\; e_2 \mapsto [v_2/x]e_1'$ — by the rule *App*
We let $e' = [v_2/x]e_1'$.

Subcase: e_1 is a value and there exists e_2' such that $e_2 \mapsto e_2'$
$e_1 = \lambda x\!:\!A.\, e_1'$ — by Lemma 1.5
$e_1\; e_2 \mapsto (\lambda x\!:\!A.\, e_1')\; e_2'$ — by the rule *Arg*
We let $e' = (\lambda x\!:\!A.\, e_1')\; e_2'$.

Subcase: there exists e_1' such that $e_1 \mapsto e_1'$
$e_1\; e_2 \mapsto e_1'\; e_2$ — by the rule *Lam*
We let $e' = e_1'\; e_2$.

Case $\dfrac{\cdot \vdash e_b : \mathsf{bool} \quad \cdot \vdash e_1 : A \quad \cdot \vdash e_2 : A}{\cdot \vdash \mathsf{if}\; e_b\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2 : A}\,\mathsf{If}$ where $e = \mathsf{if}\; e_b\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2$:
e_b is either a value or there exists e_b' such that $e_b \mapsto e_b'$
by induction hypothesis on $\cdot \vdash e_b : \mathsf{bool}$

Subcase: e_b is a value
e_b is either true or false — by Lemma 1.5
$\mathsf{if}\; e_b\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2 \mapsto e_1$ or $\mathsf{if}\; e_b\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2 \mapsto e_2$
by the rule $\mathit{If}_{\mathsf{true}}$ or $\mathit{If}_{\mathsf{false}}$
We let $e' = e_1$ or $e' = e_2$.

Subcase: there exists e_b' such that $e_b \mapsto e_b'$
$\mathsf{if}\ e_b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto \mathsf{if}\ e_b'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ by the rule *If*
We let $e' = \mathsf{if}\ e_b'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. □

1.4.2 Proof of type preservation

The proof of Theorem 1.4 is not as straightforward as the proof of Theorem 1.3 because the *If* part in the theorem contains two judgments: $\Gamma \vdash e : A$ and $e \mapsto e'$. (We have seen a similar case in the proof of Lemma **??**.) Therefore we need to decide to which judgment of $\Gamma \vdash e : A$ and $e \mapsto e'$ we apply rule induction. It turns out that the type preservation theorem is a special case in which we may apply rule induction to either judgment!

Suppose that we choose to apply rule induction to $e \mapsto e'$. Since there are six reduction rules, we need to consider (at least) six cases. The question now is: which case do we do consider first?

As a general rule of thumb, if you are proving a property that is expected to hold, the most difficult case should be the first to consider. The rationale is that eventually you have to consider the most difficult case anyway, and by considering it at an early stage of the proof, you may find a flaw in the system or identify auxiliary lemmas for the proof. Even if you discover a flaw in the system from the analysis of the most difficult case, you at least avoid considering easy cases more than once. Conversely, if you are trying to locate flaws in the system by proving a property that is not expected to hold, the easiest case should be the first to consider. The rationale is that the cheapest way to locate a flaw is by considering the easiest case in which the flaw manifests itself (although it is not as convincing as the previous rationale). The most difficult case may not even shed any light on hidden flaws in the system, thereby wasting your efforts to analyze it.

Since we wish to *prove* the type preservation theorem rather than *refute* it, we consider the most difficult case of $e \mapsto e'$ first. Intuitively the most difficult case is when $e \mapsto e'$ is proven by applying the rule App, since the substitution in it may transform an application e into a completely different form of expression, for example, a conditional construct. (The rules If_{true} and If_{false} are the easiest cases because they have no premise and e' is a subexpression of e.)

So let us consider the most difficult case in which $(\lambda x\!:\!A.\,e)\ v \mapsto [v/x]e$ holds. Our goal is to use an assumption $\Gamma \vdash (\lambda x\!:\!A.\,e)\ v : C$ to prove $\Gamma \vdash [v/x]e : C$. The typing judgment $\Gamma \vdash (\lambda x\!:\!A.\,e)\ v : C$ has the following derivation tree:

$$\dfrac{\dfrac{\Gamma, x : A \vdash e : C}{\Gamma \vdash \lambda x\!:\!A.\,e : A \!\rightarrow\! C}\ \rightarrow\!\mathsf{I} \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda x\!:\!A.\,e)\ v : C}\ \rightarrow\!\mathsf{E}$$

Therefore our new goal is to use two assumptions $\Gamma, x : A \vdash e : C$ and $\Gamma \vdash v : A$ to prove $\Gamma \vdash [v/x]e : C$. The substitution lemma below generalizes the problem:

Lemma 1.6 (Substitution). *If $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash e' : C$, then $\Gamma \vdash [e/x]e' : C$.*

The substitution lemma is similar to the type preservation theorem in that the *If* part contains two judgments. Unlike the type preservation theorem, however, we need to take great care in applying rule induction because picking up a wrong judgment makes it impossible to complete the proof!

Exercise 1.7. To which judgment do you think we have to apply rule induction in the proof of Lemma 1.6? $\Gamma \vdash e : A$ or $\Gamma, x : A \vdash e' : C$? Why?

The key observation is that $[e/x]e'$ analyzes the structure of e', not e. That is, $[e/x]e'$ searches for every occurrence of variable x in e' only to replace it by e, and thus does not even need to know the structure of e. Thus the right judgment for applying rule induction is $\Gamma, x : A \vdash e' : C$.

Proof of Lemma 1.6. By rule induction on the judgment $\Gamma, x : A \vdash e' : C$. Recall that variables in a typing context are assumed to be all distinct.

Case $\dfrac{y : C \in \Gamma, x : A}{\Gamma, x : A \vdash y : C}$ Var where $e' = y$ and $y : C \in \Gamma$:

$\Gamma \vdash y : C$ — from $y : C \in \Gamma$

$[e/x]y = y$ — from $x \neq y$

$\Gamma \vdash [e/x]y : C$ — □

Exercise 1.8. Complete the proof of Lemma 1.6. You will have to consider another case in which the rule Var is used to prove $\Gamma, x : A \vdash e' : C$.

At last, we are ready to prove the type preservation theorem. The proof proceeds by rule induction on the judgment $e \mapsto e'$. It exploits the fact that there is only one typing rule for each form of expression. For example, the only way to prove $\Gamma \vdash e_1\ e_2 : A$ is by applying the rule $\rightarrow$E. Thus the type system is said to be *syntax directed* in that the *syntactic* form of expression e in a judgment $\Gamma \vdash e : A$ decides, or *directs*, the rule to be applied. Since the syntax directedness of the type system decides a unique typing rule R for deducing $\Gamma \vdash e : A$, the premises of the rule R may be assumed to hold whenever $\Gamma \vdash e : A$ holds. For example, $\Gamma \vdash e_1\ e_2 : A$ can be proven only by applying the rule $\rightarrow$E, from which we may conclude that the two premises $\Gamma \vdash e_1 : B \rightarrow A$ and $\Gamma \vdash e_2 : B$ hold for some type B. This is called the *inversion property* which inverts the typing rule so that its conclusion justifies the use of its premises. We state the inversion property as a separate lemma.

Lemma 1.9 (Inversion). *Suppose* $\Gamma \vdash e : C$.
If $e = x$, *then* $x : C \in \Gamma$.
If $e = \lambda x\!:\!A.\, e'$, *then* $C = A \rightarrow B$ *and* $\Gamma, x : A \vdash e' : B$ *for some type* B.
If $e = e_1\ e_2$, *then* $\Gamma \vdash e_1 : A \rightarrow C$ *and* $\Gamma \vdash e_2 : A$ *for some type* A.
If $e = \mathsf{true}$, *then* $C = \mathsf{bool}$.
If $e = \mathsf{false}$, *then* $C = \mathsf{bool}$.
If $e = \mathsf{if}\ e_b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, *then* $\Gamma \vdash e_b : \mathsf{bool}$ *and* $\Gamma \vdash e_1 : C$ *and* $\Gamma \vdash e_2 : C$.

Proof. By the syntax directedness of the type system. A formal proof proceeds by rule induction on the judgment $\Gamma \vdash e : C$. □

Proof of Theorem 1.4. By rule induction on the judgment $e \mapsto e'$.

Case $\dfrac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}$ *Lam*

$\Gamma \vdash e_1\ e_2 : A$ — assumption

$\Gamma \vdash e_1 : B \rightarrow A$ and $\Gamma \vdash e_2 : B$ for some type B — by Lemma 1.9

$\Gamma \vdash e_1' : B \rightarrow A$ — by induction hypothesis on $e_1 \mapsto e_1'$ with $\Gamma \vdash e_1 : B \rightarrow A$

$\Gamma \vdash e_1'\ e_2 : A$ — from $\dfrac{\Gamma \vdash e_1' : B \rightarrow A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1'\ e_2 : A} \rightarrow\mathsf{E}$

Case $\dfrac{e_2 \mapsto e_2'}{(\lambda x\!:\!B.\, e_1')\ e_2 \mapsto (\lambda x\!:\!B.\, e_1')\ e_2'}$ *Arg*

$\Gamma \vdash (\lambda x\!:\!B.\, e_1')\ e_2 : A$ — assumption

$\Gamma \vdash \lambda x\!:\!B.\, e_1' : B \rightarrow A$ and $\Gamma \vdash e_2 : B$ — by Lemma 1.9

$\Gamma \vdash e_2' : B$ — by induction hypothesis on $e_2 \mapsto e_2'$ with $\Gamma \vdash e_2 : B$

$\Gamma \vdash (\lambda x\!:\!B.\, e_1')\ e_2' : A$ — from $\dfrac{\Gamma \vdash \lambda x\!:\!B.\, e_1' : B \rightarrow A \quad \Gamma \vdash e_2' : B}{\Gamma \vdash (\lambda x\!:\!B.\, e_1')\ e_2' : A} \rightarrow\mathsf{E}$

Case $\dfrac{}{(\lambda x\!:\!B.\, e_1')\ v \mapsto [v/x]e_1'}$ *App*

$\Gamma \vdash (\lambda x\!:\!B.\, e_1')\ v : A$ — assumption

$\Gamma \vdash \lambda x\!:\!B.\, e_1' : B \rightarrow A$ and $\Gamma \vdash v : B$ — by Lemma 1.9

$\Gamma, x : B \vdash e_1' : A$ — by Lemma 1.9 on $\Gamma \vdash \lambda x\!:\!B.\, e_1' : B \rightarrow A$

$\Gamma \vdash [v/x]e_1' : A$ — by applying Lemma 1.6 to $\Gamma \vdash v : B$ and $\Gamma, x : B \vdash e_1' : A$

Case $\dfrac{e_b \mapsto e_b'}{\textsf{if } e_b \textsf{ then } e_1 \textsf{ else } e_2 \mapsto \textsf{if } e_b' \textsf{ then } e_1 \textsf{ else } e_2}$ *If*

$\Gamma \vdash \textsf{if } e_b \textsf{ then } e_1 \textsf{ else } e_2 : A$ — assumption

$\Gamma \vdash e_b : \textsf{bool}$ and $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$ — by Lemma 1.9

$\Gamma \vdash e_b' : \textsf{bool}$ — by induction hypothesis on $e_b \mapsto e_b'$ with $\Gamma \vdash e_b : \textsf{bool}$

$\Gamma \vdash \textsf{if } e_b' \textsf{ then } e_1 \textsf{ else } e_2 : A$ — from $\dfrac{\Gamma \vdash e_b' : \textsf{bool} \quad \Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A}{\Gamma \vdash \textsf{if } e_b' \textsf{ then } e_1 \textsf{ else } e_2 : A}$ If

Case $\dfrac{}{\textsf{if true then } e_1 \textsf{ else } e_2 \mapsto e_1}$ $\mathit{If}_{\textsf{true}}$

$\Gamma \vdash \textsf{if true then } e_1 \textsf{ else } e_2 : A$ — assumption

$\Gamma \vdash \textsf{true} : \textsf{bool}$ and $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$ — by Lemma 1.9

$\Gamma \vdash e_1 : A$

Case $\dfrac{}{\textsf{if false then } e_1 \textsf{ else } e_2 \mapsto e_2}$ $\mathit{If}_{\textsf{false}}$

(Similar to the case for the rule $\mathit{If}_{\textsf{true}}$) □

1.5 Exercises

Exercise 1.10. Prove Theorem 1.4 by rule induction on the judgment $\Gamma \vdash e : A$.