Chapter 3

Simple Types

We study simple types in this chapter. We first formalize a simply-typed programming language $\mathcal{L}_0$ and then establish its type soundness, setting some machinery for development in the following chapters. We also prove a classic result based on the reducibility method (Tait 1967) that simply-typed $\lambda$-calculus is strongly normalizing, namely, there is no infinite $\beta$-reduction sequence starting from any simply-typed $\lambda$-term. This proof method is to be employed later for building a theorem-proving system in which proofs are constructed as total functions.

3.1 A Simply-Typed Programming Language

We present in Figure 3 the syntax for a simply-typed language $\mathcal{L}_0$. We use $\delta$ for base types, which include $\text{int}$ for integers and $\text{bool}$ for booleans. We use $c$ for a constant, which is either a constant constructor $cc$ or a constant function $cf$. For instance, we have $\text{true}$ and $\text{false}$ for boolean constants, and $0, -1, 1, -2, 2, \ldots$ for integer constants, and functions such as $+$ and $-$ on integers. We can simply write $c$ for the expression $c()$ if the constant $c$ is of arity 0. We also have lam-variables $x$ and fix-variables $f$, and may use $xf$ for either a lam-variable or a fix-variable. Note that each lam-variable following $\text{lam}$ is a bound variable and so is each fix-variable following $\text{fix}$. We use $v$ for values, which are a special form of expressions. Note that a lam-variable is a value but a fix-variable is not. We use $\vec{e}$ for a (possibly empty) sequence of expressions, and this notation may also be applied to variables, values, etc.

| types          | $T ::= \delta | T_1 * T_2 | T_1 \rightarrow T_2$ |
| expressions    | $e ::= x | c(\vec{e}) | \text{if}(e_0, e_1, e_2) | \langle e_1, e_2 \rangle | \text{fst}(e) | \text{snd}(e) | \text{lam}x.e | \text{app}(e_1, e_2) | \text{fix}f.e$ |
| values         | $v ::= x | cc(\vec{v}) | \langle v_1, v_2 \rangle | \text{lam}x.e$ |
| typing contexts| $\Gamma ::= \emptyset | \Gamma, x : T$ |
| evaluation contexts | $E ::= [] | c(\vec{v}, E, \vec{e}) | \text{if}(E, e_1, e_2) | \langle E, e \rangle | \langle v, E \rangle | \text{fst}(E) | \text{snd}(E) | \text{app}(E, e) | \text{app}(v, E)$ |

Figure 3.1: The syntax for $\mathcal{L}_0$
Various functions on \( \lambda \)-terms can be readily defined on expressions in \( \mathcal{L}_0 \) as well. For instance, it should be obvious how \( \text{FV}(\cdot) \) can be defined in \( \mathcal{L}_0 \). In particular, we have \( \text{FV}(\text{lam } x.e) = \text{FV}(e) \setminus \{x\} \) \( \text{FV}(\text{fix } f.e) = \text{FV}(e) \setminus \{f\} \). Also, it should be clear how \( \alpha \)-renaming can be performed on expressions in \( \mathcal{L}_0 \).

We are to present the dynamic semantics of \( \mathcal{L}_0 \) based on redexes and evaluation contexts. A redex \( \mathcal{L}_0 \) is a special form of an expression, which can be reduced to another expression referred to as a reduct of the redex. In \( \mathcal{L}_0 \), each redex has at least one reduct.

**Definition 3.1.1 (Redexes)** The redexes in \( \mathcal{L}_0 \) and their reducts are defined below.

- \( \text{cf}(\vec{v}) \) is a redex, and each defined value of \( \text{cf}(\vec{v}) \) is a reduct of \( \text{cf}(\vec{v}) \).
- \( \text{if}(\text{true}, e_1, e_2) \) is a redex, and its reduct is \( e_1 \).
- \( \text{if}(\text{false}, e_1, e_2) \) is a redex, and its reduct is \( e_2 \).
- \( \text{fst}(\langle v_1, v_2 \rangle) \) is a redex, and its reduct is \( v_1 \).
- \( \text{snd}(\langle v_1, v_2 \rangle) \) is a redex, and its reduct is \( v_2 \).
- \( \text{app}(\text{lam } x.e, v) \) is a redex, and its reduct is \( e[x := v] \).
- \( \text{fix } f.e \) is a redex, and its reduct is \( e[f := \text{fix } f.e] \).

For instance, if we assume that \( + \) represents the usual addition function on integers, then \( 1 + 2 \) is a redex and \( 3 \) is the only reduct of \( 1 + 2 \). More interestingly, we may also assume the existence of a nullary constant function \( \text{random} \) such that \( \text{random}() \) is a redex and every natural number is a reduct of \( \text{random}() \).

**Definition 3.1.2 (Evaluation)** We use \( \rightarrow \) for the binary evaluation relation on expressions. Given \( e_1 \) and \( e_2, e_1 \rightarrow e_2 \) holds if \( e_1 = E[e] \) and \( e_2 = E[e'] \) for some evaluation context \( E \), redex \( e \) and a reduct \( e' \) of \( e \). We use \( \rightarrow^* \) for the reflexive and transitive closure of \( \rightarrow \).

There are clearly (closed) non-value expressions in \( \mathcal{L}_0 \) that can not be further evaluated, and we refer to as such expressions as being stuck. For instance, expressions like \( \text{fst}(0) \) and \( \text{app}(1, \text{true}) \) are stuck.

Given a program \( e \), that is, a closed expression, we expect to avoid the scenario where evaluating the program may reach a stuck expression. To address this issue, we set out to enforce a type discipline in \( \mathcal{L}_0 \) so that for each well-typed program, that is, program constructed following the type discipline, evaluating the program can never lead to a stuck expression.

A typing judgment in \( \mathcal{L}_0 \) is of the form \( \Gamma \vdash e : T \), meaning that \( e \) can be assigned the type \( T \) under the typing context \( \Gamma \). The rules for deriving such judgments are given in Figure 3.2. In \( \mathcal{L}_0 \), each constant constructor \( cc \) is given a type of the form \( (T_1, \ldots, T_n) \rightarrow \delta \), and we say that \( cc \) is associated with \( \delta \). For instance, \( \text{true} \) and \( \text{false} \) are given the type \( () \rightarrow \text{bool} \), and each integer constant is given the type \( () \rightarrow \text{int} \).

**Lemma 3.1.3 (Canonical Forms)** Assume that \( \emptyset \vdash v : T \) is derivable.
3.1. A SIMPLY-TYPED PROGRAMMING LANGUAGE

\[ \Gamma(x) = T \]
\[ \Gamma \vdash x : T \] (ty-var)

\[ \vdash c : (T_1, \ldots, T_n) \rightarrow T \quad \Gamma \vdash e_i : T_i \text{ for } 1 \leq i \leq n \]
\[ \Gamma \vdash c(e_1, \ldots, e_n) : T \] (ty-cst)

\[ \Gamma \vdash e_0 : \text{bool} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T \]
\[ \Gamma \vdash \text{if}(e_0, e_1, e_2) : T \] (ty-if)

\[ \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \]
\[ \Gamma \vdash (e_1, e_2) : T_1 \times T_2 \] (ty-tup)

\[ \Gamma \vdash e : T_1 \times T_2 \]
\[ \Gamma \vdash \text{fst}(e) : T_1 \] (ty-fst)
\[ \Gamma \vdash e : T_1 \times T_2 \]
\[ \Gamma \vdash \text{snd}(e) : T_2 \] (ty-snd)

\[ \Gamma, x : T_1 \vdash e : T_2 \]
\[ \Gamma \vdash \text{lam } x.e : T_1 \rightarrow T_2 \] (ty-lam)

\[ \Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 : T_1 \]
\[ \Gamma \vdash \text{app}(e_1, e_2) : T_2 \] (ty-app)

\[ \Gamma, f : T \vdash e : T \]
\[ \Gamma \vdash \text{fix } f.e : T \] (ty-fix)

Figure 3.2: The typing rules for expressions in \( L_0 \)

- If \( T = \delta \) for some base type \( \delta \), then \( v \) is of the form \( \text{cc}(\vec{v}) \), where \( \text{cc} \) is a constructor associated with \( \delta \).

- If \( T = T_1 \times T_2 \) for some types \( T_1 \) and \( T_2 \), then \( v \) is of the form \( \langle v_1, v_2 \rangle \).

- If \( T = T_1 \rightarrow T_2 \) for some types \( T_1 \) and \( T_2 \), then \( v \) is of the form \( \text{lam } x.e \)

**Proof** By an inspection of the typing rules in Figure 3.2.

We use \( \theta \) for substitutions in \( L_0 \), which are finite mappings from variables to expressions. Given a substitution \( \theta \) and an expression, \( e[\theta] \) is the result from applying \( \theta \) to \( e \), which can be formally defined as is done in Chapter 2. In particular, when applying \( \theta \) to \( \text{lam } x.e \), we may need to first \( \alpha \)-rename \( \text{lam } x.e \) to some \( \text{lam } x'.e' \) so that \( x' \) does not occur in \( \text{vars}(\theta) \) and then obtain \( \text{lam } x'.e'[\theta] \). Similarly, when applying \( \theta \) to \( \text{fix } f.e \), we may need to \( \alpha \)-rename the bound variable \( f \).

**Lemma 3.1.4 (Substitution)** Assume that \( \Gamma_0, \Gamma \vdash e : T \) is derivable and \( \Gamma_0 \vdash \theta : \Gamma \) holds. Then \( \Gamma_0 \vdash e[\theta] : T \) is also derivable.

**Proof** We proceed by induction on the height of the typing derivation \( D \) of \( \Gamma_0, \Gamma \vdash e : T \).

- \( e \) is some variable \( x \). Then \( x \in \text{dom}(\Gamma_0, \Gamma) \). If \( x \in \text{dom}(\Gamma) \), then \( \Gamma_0 \vdash \theta : \Gamma \) implies that \( \Gamma_0 \vdash e[\theta] : \Gamma(x) \) is derivable since \( e[\theta] = \theta(x) \). If \( x \notin \text{dom}(\Gamma) \), then \( x \in \text{dom}(\Gamma_0) \) and thus \( \Gamma_0 \vdash x : T \) is derivable.
• \( e \) is of the form \( (e_1, e_2) \). Then \( D \) must be of the following form:

\[
D_1 ::: \Gamma_0, \Gamma \vdash e_1 : T_1 \quad D_2 ::: \Gamma_0, \Gamma \vdash e_2 : T_2
\]

\[
\Gamma_0, \Gamma \vdash e : T
\]

where \( T = T_1 \ast T_2 \). By induction hypotheses on \( D_1 \) and \( D_2 \), both \( \Gamma_0 \vdash e_1[\theta] : T_1 \) and \( \Gamma_0 \vdash e_2[\theta] : T_2 \) are derivable. Hence, \( \Gamma_0 \vdash (e_1[\theta], e_2[\theta]) : T \) is derivable. Note that \( e[\theta] = (e_1[\theta], e_2[\theta]) \), and we are done.

• \( e \) is of the form \( \text{lam} \ x.e_1 \). Then \( D \) must be of the following form:

\[
D_1 ::: \Gamma_0, \Gamma, x : T_1 \vdash e_1 : T_2
\]

\[
\Gamma_0, \Gamma \vdash \text{lam} \ x.e_1 : T
\]

We may assume that \( x \) is distinct from each variable in \( \Gamma \). Clearly, we can construct a derivation \( D' \) of \( \Gamma_0, x : T_1, \Gamma \vdash e_1 : T_2 \) that is of the same height as \( D_1 \). By induction hypothesis on \( D' \), we can derive \( \Gamma_0, x : T_1 \vdash e_1[\theta] : T_2 \), which leads to a derivation of \( \Gamma_0 \vdash \text{lam} \ x.e_1[\theta] : T_2 \). Note that \( e[\theta] = \text{lam} \ x.e_1[\theta] \), and we are done.

The rest of the cases are omitted, which can all be handled similarly.

**Lemma 3.1.5** Assume that \( \Gamma \vdash e : T \) is derivable. If \( e \) is a redex and \( e' \) is a reduct of \( e \), then \( \Gamma \vdash e' : T \) is also derivable.

**Proof** As an exercise.

A typing judgment for assigning types to evaluation contexts in \( L_0 \) is of the form \( \Gamma \vdash E : T_0/T \), meaning that \( E \) can be assigned the type \( T \) under \( \Gamma \) if the hole \([-] \) in \( E \) is given the type \( T_0 \). The rules for deriving such judgments are given in Figure 3.3. As these rules can be easily constructed by studying the typing rules in Figure 3.2, we may omit presenting such rules in the future.

**Lemma 3.1.6** If both \( \Gamma \vdash E : T_0/T \) and \( \Gamma \vdash e : T_0 \) are derivable, then \( \Gamma \vdash E[e] : T \) is also derivable.

**Proof** As an exercise.

**Lemma 3.1.7** If \( \Gamma \vdash E[e] : T \) is derivable, then there exists a type \( T_0 \) such that both \( \Gamma \vdash E : T_0/T \) and \( \Gamma \vdash e : T_0 \) are both derivable.

**Proof** As an exercise.

**Theorem 3.1.8 (Subject Reduction)** Assume that \( \Gamma \vdash e_1 : T \) is derivable and \( e_1 \rightarrow e_2 \) holds. Then \( \Gamma \vdash e_2 : T \) is also derivable.

**Proof** Assume that \( e_1 = E[e] \) for some evaluation context \( E \) and redex \( e \), and \( e_2 = E[e'] \) for some reduct \( e' \) of \( e \). By Lemma 3.1.7, there exists a type \( T_0 \) such that \( \Gamma \vdash E : T_0 \rightarrow T \) and \( \Gamma \vdash e : T_0 \) are derivable. By Lemma 3.1.5, \( \Gamma \vdash e' : T_0 \) is derivable, and by Lemma 3.1.6, \( \Gamma \vdash e_2 : T \) is derivable since \( e_2 = E[e'] \).

**Lemma 3.1.9** Assume that \( \emptyset \vdash e : T \) is derivable. If \( e \) is not a value, then \( e = E[e'] \) for some evaluation context \( E \) and redex \( e' \).
### 3.1. A SIMPLY-TYPED PROGRAMMING LANGUAGE

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \Gamma \vdash \emptyset : T_0/T_0 ]</td>
<td>(tc-id)</td>
</tr>
<tr>
<td>[ \vdash c : (T_1, \ldots, T_n) \rightarrow T \quad \Gamma \vdash E : T_0/T_i ]</td>
<td>(tc-cst)</td>
</tr>
<tr>
<td>[ \Gamma \vdash v_k : T_k \text{ for } 1 \leq k &lt; i \quad \Gamma \vdash e_k : T_k \text{ for } i &lt; k \leq n ]</td>
<td></td>
</tr>
<tr>
<td>[ \Gamma \vdash c(v_1, \ldots, v_{i-1}, E, e_{i+1}, \ldots, e_n) : T_0/T ]</td>
<td></td>
</tr>
<tr>
<td>[ \Gamma \vdash E : T_0/\text{bool} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T ]</td>
<td>(tc-if)</td>
</tr>
<tr>
<td>[ \Gamma \vdash \text{if}(E, e_1, e_2) : T_0/T ]</td>
<td>(tc-tup-1)</td>
</tr>
<tr>
<td>[ \Gamma \vdash E : T_0/T_1 \quad \Gamma \vdash e : T_2 ]</td>
<td></td>
</tr>
<tr>
<td>[ \Gamma \vdash \langle E, e \rangle : T_0/T_1 \ast T_2 ]</td>
<td>(tc-tup-2)</td>
</tr>
<tr>
<td>[ \Gamma \vdash E : T_0/T_1 \ast T_2 ]</td>
<td></td>
</tr>
<tr>
<td>[ \Gamma \vdash \text{fst}(E) : T_0/T_1 ]</td>
<td>(tc-fst)</td>
</tr>
<tr>
<td>[ \Gamma \vdash E : T_0/T_1 \ast T_2 ]</td>
<td>(tc-snd)</td>
</tr>
<tr>
<td>[ \Gamma \vdash E : T_0/T_1 \ast T_2 ]</td>
<td>(tc-app)</td>
</tr>
<tr>
<td>[ \Gamma \vdash e : T_1 \rightarrow T_2 \quad \Gamma \vdash E : T_0/T_1 ]</td>
<td></td>
</tr>
<tr>
<td>[ \Gamma \vdash \text{app}(e, E) : T_0/T_2 ]</td>
<td>(tc-app)</td>
</tr>
<tr>
<td>[ \Gamma \vdash E : T_0/T_1 \rightarrow T_2 \quad \Gamma \vdash e : T_1 ]</td>
<td></td>
</tr>
<tr>
<td>[ \Gamma \vdash \text{app}(E, e) : T_0/T_2 ]</td>
<td>(tc-app)</td>
</tr>
</tbody>
</table>

Figure 3.3: The typing rules for evaluation contexts in $\mathcal{L}_0$
\textbf{Proof} We proceed by structural induction on the typing derivation $D$ of $\emptyset \vdash e : T$.

- The last rule applied in $D$ is (\textsc{ty-cst}). Then $D$ is of the following form:

$$
\frac{
\vdash c : (T_1, \ldots, T_n) \rightarrow T \\
D_i :: \emptyset \vdash e_i : T_i \text{ for } 1 \leq i \leq n
}{\emptyset \vdash c(e_1, \ldots, e_n) : T} \quad \text{(ty-cst)}
$$

where $e = c(e_1, \ldots, e_n)$. We have two subcases.

  - There exists $e_i$ for some $1 \leq i \leq n$ that is not a value but $e_j$ are values for all $1 \leq j < i$. By induction hypothesis on $D_i$, $e_i = E_1[e']$ for some evaluation context $E_1$ and redex $e'$. Let $E = c(e_1, \ldots, e_{i-1}, E_1, e_{i+1}, \ldots, e_n)$, and we are done.

  - All $e_i$ are values for $1 \leq i \leq n$. Since $e$ is not a value, $c$ must be a constant function and thus $e$ is a redex. Let $E$ be $[]$ and $e'$ be $e$, and we are done.

- The last rule applied in $D$ is (\textsc{ty-if}). Then $D$ is of the following form:

$$
\frac{
D_0 :: \emptyset \vdash e_0 : \text{bool} \\
D_1 :: \emptyset \vdash e_1 : T_1 \\
D_2 :: \emptyset \vdash e_2 : T_2
}{\emptyset \vdash \text{if}(e_0, e_1, e_2) : T} \quad \text{(ty-if)}
$$

where $e = \text{if}(e_0, e_1, e_2)$. We have two subcases.

  - $e_0$ is not a value. Then by induction hypothesis on $D_0$, $e_0 = E_1[e']$ for some evaluation context $E_1$ and redex $e'$. Let $E = \text{if}(E_1, e_1, e_2)$, and we are done.

  - $e_0$ is a value. By Lemma 3.1.3, $e_0$ is either true or false. Hence, $e$ is a redex. Let $E$ be $[]$ and $e'$, and we are done.

- The last rule applied in $D$ is (\textsc{ty-fst}). Then $D$ is of the following form:

$$
\frac{
D_0 :: \emptyset \vdash e_0 : T_1 * T_2
}{\emptyset \vdash \text{fst}(e_0) : T_1} \quad \text{(ty-fst)}
$$

where $e = \text{fst}(e_0)$ and $T = T_1$. We have two subcases.

  - $e_0$ is not a value. By induction hypothesis on $D_0$, $e_0 = E_1[e']$ for some evaluation context $E_1$ and redex $e'$. Let $E = \text{fst}(E_1)$, and we are done.

  - $e_0$ is a value. By Lemma 3.1.3, $e_0$ is of the form $\langle v_1, v_2 \rangle$. So $e$ is a redex. Let $E$ be $[]$ and $e'$, and we are done.

- The last rule applied in $D$ is (\textsc{ty-snd}). Then this case is symmetric to the previous one.

- The last rule applied in $D$ is (\textsc{ty-app}). Then $D$ is of the following form:

$$
\frac{
D_1 :: \emptyset \vdash e_1 : T_1 \rightarrow T_2 \\
D_2 :: \emptyset \vdash e_2 : T_1
}{\emptyset \vdash \text{app}(e_1, e_2) : T_2} \quad \text{(ty-app)}
$$

where $e = \text{app}(e_1, e_2)$ and $T = T_2$. We have a few subcases.
3.2. P A T T E R N M A T C H I N G

- $e_1$ is not a value. By induction hypothesis on $D_1$, $e_1 = E_1[e^r]$. Let $E = \text{app}(E_1, e_2)$ and we are done.

- $e_1$ is a value but $e_2$ is not a value. By induction hypothesis on $D_2$, $e_2 = E_1[e^r]$. Let $E = \text{app}(e_1, E_1)$ and we are done.

- $e_1$ and $e_2$ are values. By Lemma 3.1.3, $e_1$ is of the form $\text{lam} \ x.e_{10}$. Hence, $e$ is a redex. Let $E = []$ and $e^r$, and we are done.

- The last rule applied in $D$ is (ty-fix). Then $D$ is of the following form:

\[
\frac{\emptyset, f : T \vdash e_0 : T}{\emptyset \vdash \text{fix} \ f. e_0 : T} \quad \text{(ty-fix)}
\]

where $e = \text{fix} \ f. e_0$. Then $e$ is a redex. Let $E = []$ and $e^r$, and we are done.

We conclude the proof as all the possible cases are covered.

\[\square\]

Theorem 3.1.10 (Progress) Assume that $\emptyset \vdash e : T$ is derivable. Then either $e$ is value or $e \rightarrow e'$ for some $e'$.

Proof Assume that $e$ is not a value. By Lemma 3.1.9, $e = E[e^r]$ for some evaluation context $E$ and redex $e^r$. Hence, $e \rightarrow E[e^c]$ holds for each $e^c$ that is a reduct of $e^r$.

By Theorem 3.1.8 and Theorem 3.1.10, it is clear that for every well-typed (closed) expression $e$ in $L_0$, an evaluation starting from $e$ either terminates with a value or continues forever. In particular, $e$ can never be evaluated to a non-value expression that cannot be further evaluated. This is phrased by Robin Milner in the slogan: a well-typed program can never go wrong.

3.2 Pattern Matching

3.3 Simply-Typed Lambda-Calculus

The types in simply-typed lambda-calculus are given as follows, where we use $\delta$ for some base types.

\[
\text{types} \quad T ::= \delta \mid T_1 \rightarrow T_2
\]

These types are often referred to as simply types. For each type $T$, we assume the existence of a denumerable set of (typed) variables $x_T^1, x_T^2, \ldots$ and use $x_T^1, y_T, z_T^1$ to ranges over this set. We may write $x$ to mean $x_T$ for some $T$.

Definition 3.3.1 The simply-typed $\lambda$-terms are defined as follows.

- A typed variable $x_T^1$ is a term of type $T$.
- $\lambda x_{T_1}. t$ is a term of type $T_1 \rightarrow T_2$ if $t$ is a term of type $T_2$.
- $t_1(t_2)$ is a term of type $T_2$ if $t_1$ and $t_2$ are terms of types $T_1 \rightarrow T_2$ and $T_1$, respectively.
We say that a term is neutral if it is a variable or an application. We may write $t^T$ to indicate that $t$ is a term of type $T$.

**Definition 3.3.2 (Reducibility)** Given a term $t$ of type $T$, $t$ is reducible of type $T$ if $\mu(t) < \infty$ and

- $T = \delta$ for some $\delta$, or
- $T = T_1 \rightarrow T_2$ and for every term $t_1$ that is reducible of type $T_1$, $t_0[x := t_1]$ is reducible of type $T_2$ whenever $t \rightarrow^{\ast}_{\beta} \lambda x. t_0$ holds.

Given a term of type $T$, we say that $t$ is reducible if $t$ is reducible of type $T$.

Clearly, the notion of being reducible of type $T$ is well-defined based on structural induction on $T$. This notion is often referred to as reducibility of type $T$. In Definition 3.3.2, the notion of reducibility is presented in a style we refer to as *introduction-major*. This is in contrast to the definition given in (Girard, Lafont, and Taylor 1989), which is of a style we refer to as *elimination-major*.

**Lemma 3.3.3 (Forward Property)** Assume that $t$ is reducible and $t \rightarrow^{\ast}_{\beta} t'$. Then $t'$ is also reducible.

**Proof** The lemma follows from the definition of reducibility immediately.

**Lemma 3.3.4 (Backward Property)** Assume that $t$ is neutral. If $t'$ is reducible whenever $t \rightarrow^{\ast}_{\beta} t'$ holds, then $t$ is also reducible.

**Proof** Assume that $t$ is of type $T$. Clearly, $\mu(t)$ is finite as $\mu(t') < \infty$ whenever $t \rightarrow^{\ast}_{\beta} t'$ holds. If $T$ is some $\delta$, we are done. Otherwise, assume $T = T_1 \rightarrow T_2$. If $t \rightarrow^{\ast}_{\beta} \lambda x. t_0$, then there must be $t'$ such that $t \rightarrow^{\ast}_{\beta} t'$ and $t' \rightarrow^{\ast}_{\beta} \lambda x. t_0$. Since $t'$'s is reducible, $t_0[x := t_1]$ is reducible for every $t_1$ that is reducible of type $T_1$. By definition, $t$ is reducible of type $T$.

**Corollary 3.3.5** Every variable $x^T$ is reducible of type $T$.

**Proof** By Lemma 3.3.4 immediately.

**Lemma 3.3.6** Let $t = \lambda x^{T_1}. t_0$. If $t_0[x := t_1]$ is reducible for every $t_1$ that is reducible of type $T_1$, then $t$ is reducible.

**Proof** Assume $t \rightarrow^{\ast}_{\beta} \lambda x. t_0$. Then we have $t_0 \rightarrow^{\ast}_{\beta} t_0'$, which implies $t_0[x := t_1] \rightarrow^{\ast}_{\beta} t_0'[x := t_1]$. By Lemma 3.3.3, $t_0'[x := t_1]$ is reducible. By definition, $t$ is reducible.

**Lemma 3.3.7** Assume that $t_1$ and $t_2$ are reducible of types $T_1 \rightarrow T_2$ and $T_2$, respectively. Then $t_1(t_2)$ is reducible of type $T_2$.

**Proof** By definition, $\mu(t_i) < \infty$ for $i = 1, 2$. Let $t = t_1(t_2)$, and we proceed by induction on $\mu(t_1) + \mu(t_2)$. Assume $t \rightarrow^{\ast}_{\beta} t'$, and we have the following cases.

- $t_1 \rightarrow^{\ast}_{\beta} t_1'$ and $t' = t_1'(t_2)$. By Lemma 3.3.3, $t_1'$ is reducible. Note that $\mu(t_1') + \mu(t_2) < \mu(t_1) + \mu(t_2)$. By induction hypothesis, $t'$ is reducible.
• $t_2 \rightarrow_\beta t'_2$ and $t' = t_1(t'_2)$. This case is similar to the previous one.

• $t_1 = \lambda x.t_{10}$ and $t' = t_{10}[x := t_2]$. By definition, $t_1$ and $t_2$ being reducible implies that $t'$ is reducible.

Therefore, $t'$ is reducible whenever $t \rightarrow_\beta t'$ holds. By Lemma 3.3.4, $t$ is reducible. ■

**Theorem 3.3.8** Every simply-typed \(\lambda\)-term $t$ is strongly normalizing, that is, $\mu(t) < \infty$.

**Proof** We say that a substitution $\theta$ is reducible if it maps each variable $x^T$ in its domain to a term reducible of type $T$. Assume that $t$ is of type $T_0$. We prove that $t[\theta]$ is reducible of type $T_0$ whenever $\theta$ is reducible. The proof proceeds by structural induction on $t$.

• Assume $t = x$. If $x \in \text{dom}(\theta)$, then $t[\theta] = \theta(x)$, which is reducible since $\theta$ is reducible. If $x \notin \text{dom}(\theta)$, then $t[\theta] = x$, which is reducible by Corollary 3.3.5.

• Assume $t = \lambda x^{T_1}.t_1$, where $t_1$ is a term of type $T_2$. We can choose $x^{T_1} \notin \text{vars}(\theta)$ so that $t[\theta] = \lambda x^{T_1}.t_1[\theta]$ holds. Let $t_2$ be a reducible term of type $T_1$. Note that $t_1[\theta][x^{T_1} := t_2] = t_1[\theta']$, where $\theta' = \theta[x := t_2]$ is reducible. By induction hypothesis on $t_1$, $t_1[\theta][x^{T_1} := t_2]$ is reducible. By Lemma 3.3.6, $t[\theta]$ is reducible.

• Assume $t = t_1(t_2)$. By induction hypothesis, $t_i[\theta]$ is reducible for $i = 1, 2$. By Lemma 3.3.7, $t_1[\theta](t_2[\theta])$ is reducible. Note $t[\theta] = t_1[\theta](t_2[\theta])$, and we are done.

As all the cases are covered, we conclude that $t[\theta]$ is reducible whenever $\theta$ is reducible. Let $\theta = []$, and we know that $t$ is reducible. By the definition of reducibility, $t$ is strongly normalizing. ■

### 3.4 Exercises

**Exercise 1** Extend the simply-typed \(\lambda\)-calculus with product types. Prove SN.

**Exercise 2** Extend the simply-typed \(\lambda\)-calculus with sum types. Prove SN.