

BU CAS CS 520

Principles of Programming Languages

Lecture Notes

Hongwei Xi
Computer Science Department, Boston University
111 Cummington Street, Boston, MA 02215

Chapter 1

Simply-Typed Lambda-Calculus

1.1 Syntax

constants	c	::=	$\mathbf{true} \mid \mathbf{false} \mid 0 \mid 1 \mid -1 \mid \dots$
operators	op	::=	$+ \mid - \mid * \mid / \mid \dots$
terms	t	::=	$c \mid x \mid \mathbf{if} \ t_0 \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2 \mid op(t) \mid \lambda x : T.t \mid t_1(t_2)$
values	v	::=	$c \mid \lambda x : T.t$
types	T	::=	$Bool \mid Int \mid T_1 \rightarrow T_2$
contexts	Γ	::=	$\emptyset \mid \Gamma, x : T$

1.2 Static Semantics

$$\frac{c \in \{\mathbf{true}, \mathbf{false}\}}{\Gamma \vdash c : Bool} \text{ (T-Bool)}$$
$$\frac{c \in \{0, 1, -1, \dots\}}{\Gamma \vdash c : Int} \text{ (T-Int)}$$
$$\frac{\Gamma \vdash t_0 : Bool \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \mathbf{if} \ t_0 \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2 : T} \text{ (T-If)}$$
$$\frac{\Sigma(op) = T_1 \rightarrow T_2 \quad \Gamma \vdash t : T_1}{\Gamma \vdash op(t) : T_2} \text{ (T-Op)}$$
$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \text{ (T-Var)}$$
$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \rightarrow T_2} \text{ (T-Abs)}$$
$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1(t_2) : T_2} \text{ (T-App)}$$

1.3 Dynamic Semantics

$$\begin{array}{c}
\frac{t_0 \rightarrow t'_0}{\text{if } t_0 \text{ then } t_1 \text{ else } t_2 \rightarrow \text{if } t'_0 \text{ then } t_1 \text{ else } t_2} \text{ (E-If)} \\
\frac{}{\text{if true then } t_1 \text{ else } t_2 \rightarrow t_1} \text{ (E-IfTrue)} \\
\frac{}{\text{if false then } t_1 \text{ else } t_2 \rightarrow t_2} \text{ (E-IfFalse)} \\
\frac{t \rightarrow t'}{op(t) \rightarrow op(t')} \text{ (E-Op)} \\
\frac{op(v) = v'}{op(v) \rightarrow v'} \text{ (E-OpVal)} \\
\frac{t_1 \rightarrow t'_1}{t_1(t_2) \rightarrow t'_1(t_2)} \text{ (E-App1)} \\
\frac{t_2 \rightarrow t'_2}{v_1(t_2) \rightarrow v_1(t'_2)} \text{ (E-App2)} \\
\frac{}{(\lambda x.t_1)(v_2) \mapsto [x \mapsto v_2]t_1} \text{ (E-AppAbs)}
\end{array}$$

1.4 Properties of Typing

Lemma 1 (*Inversion*) Given a typing derivation $\mathcal{D} :: \Gamma \vdash t : T$, the last applied typing rule in \mathcal{D} is uniquely determined by the syntactic structure of t .

Lemma 2 (*Canonical Forms*)

1. If v is a closed value of the type $Bool$, then v is either `true` or `false`.
2. If v is a closed value of the type Int , then $v \in \{0, 1, -1, \dots\}$ holds.
3. If v is a closed value of the type $T_1 \rightarrow T_2$, then v is of the form $\lambda x : T_1.t$.

Theorem 3 (*Progress*) Suppose that t is a closed and well-typed terms, that is, there exists a derivation $\mathcal{D} :: \emptyset \vdash t : T$. Then either t is a value or $t \rightarrow t'$ holds for some term t' .

Proof We proceed by structural induction on \mathcal{D} . Clearly, we only need to handle the cases where t is not a value.

- \mathcal{D} is of the following form,

$$\frac{\mathcal{D}_0 :: \Gamma \vdash t_0 : Bool \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \text{if } t_0 \text{ then } t_1 \text{ else } t_2 : T} \text{ (T-If)}$$

where $t = \text{if } t_0 \text{ then } t_1 \text{ else } t_2$.

- t_0 is a value. Then t_0 is either `true` or `false` by Lemma 2. Hence, $t \rightarrow t_1$ or $t \rightarrow t_2$ according to t being `true` or `false`.

- t_0 is not a value. By induction hypothesis on \mathcal{D}_0 , $t_0 \rightarrow t'_0$ for some t'_0 . Hence, $t \rightarrow \mathbf{if } t'_0 \mathbf{ then } t_1 \mathbf{ else } t_2$.

- \mathcal{D} is of the following form,

$$\frac{\Sigma(op) = T_1 \rightarrow T_2 \quad \Gamma \vdash t_1 : T_1}{\mathcal{D}_1 :: \Gamma \vdash op(t_1) : T_2} \text{ (T-Op)}$$

where $t = op(t_1)$ and $T = T_2$.

- t_1 is not a value. Then $t_1 \rightarrow t'_1$ by induction hypothesis on \mathcal{D}_1 . Hence, $t \rightarrow op(t'_1)$.
- t_1 is a value. Then $t \rightarrow v$, where v is the value of $op(t_1)$.

- \mathcal{D} is of the following form,

$$\frac{\mathcal{D}_1 :: \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \mathcal{D}_2 :: \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1(t_2) : T_2} \text{ (T-App)}$$

where $t = t_1(t_2)$ and $T = T_1$.

- t_1 is not a value. Then by induction hypothesis on \mathcal{D}_1 , $t_1 \rightarrow t'_1$ for some t'_1 . Hence, $t \rightarrow t'_1(t_2)$.
- t_1 is a value. Then by induction hypothesis on \mathcal{D}_2 , $t_2 \rightarrow t'_2$ for some t'_2 . Hence, $t \rightarrow t_1(t'_2)$.
- t_1 and t_2 are values. Then by Lemma 2, t_1 is of the form $\lambda x : T.t'_1$. Hence, $t \rightarrow [x \mapsto t_2]t'_1$. ■

Lemma 4 (*Substitution*) *If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.*

Theorem 5 (*Subject Reduction*) *If $\mathcal{D} :: \Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$ is derivable.*

Proof We proceed by structural induction on \mathcal{D} . Note that t is not a value since $t \rightarrow t'$ holds.

- \mathcal{D} is of the following form,

$$\frac{\mathcal{D}_0 :: \Gamma \vdash t_0 : Bool \quad \mathcal{D}_1 :: \Gamma \vdash t_1 : T \quad \mathcal{D}_2 :: \Gamma \vdash t_2 : T}{\Gamma \vdash \mathbf{if } t_0 \mathbf{ then } t_1 \mathbf{ else } t_2 : T} \text{ (T-If)}$$

where $t = \mathbf{if } t_0 \mathbf{ then } t_1 \mathbf{ else } t_2$.

- $t_0 \rightarrow t'_0$ and $t \rightarrow t' = \mathbf{if } t'_0 \mathbf{ then } t_1 \mathbf{ else } t_2$. By induction hypothesis on \mathcal{D}_0 , we have derivation $\mathcal{D}'_0 :: \Gamma \vdash t'_0 : Bool$. Therefore, $\Gamma \vdash t' : T$ can be derived as follows.

$$\frac{\mathcal{D}'_0 :: \Gamma \vdash t'_0 : Bool \quad \mathcal{D}_1 :: \Gamma \vdash t_1 : T \quad \mathcal{D}_2 :: \Gamma \vdash t_2 : T}{\Gamma \vdash \mathbf{if } t'_0 \mathbf{ then } t_1 \mathbf{ else } t_2 : T} \text{ (T-If)}$$

- $t_0 = \mathbf{true}$ and $t \rightarrow t' = t_1$. Note $\mathcal{D}_1 :: \Gamma \vdash t_1 : T$ and we are done.
- $t_0 = \mathbf{false}$ and $t \rightarrow t' = t_2$. Note $\mathcal{D}_2 :: \Gamma \vdash t_2 : T$ and we are done.

- \mathcal{D} is of the following form,

$$\frac{\Sigma(op) = T_1 \rightarrow T_2 \quad \Gamma \vdash t_1 : T_1}{\mathcal{D}_1 :: \Gamma \vdash op(t_1) : T_2} \text{ (T-Op)}$$

where $t = op(t_1)$ and $T = T_2$.

- $t_1 \rightarrow t'_1$ and $t \rightarrow t' = op(t'_1)$. By induction hypothesis on \mathcal{D}_1 , we have $\mathcal{D}'_1 :: \Gamma \vdash t'_1 : T_1$. Hence, $\Gamma \vdash t' : T$ can be derived as follows.

$$\frac{\Sigma(op) = T_1 \rightarrow T_2 \quad \Gamma \vdash t_1 : T_1}{\mathcal{D}'_1 :: \Gamma \vdash op(t_1) : T_2} \text{ (T-Op)}$$

- t_1 is a value and $t \rightarrow t' = v$ for the value v of $op(t_1)$. Given the type of op , v must have the type T_2 . Hence, $\Gamma \vdash t' : T$ is derivable.

- \mathcal{D} is of the following form,

$$\frac{\mathcal{D}_1 :: \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \mathcal{D}_2 :: \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1(t_2) : T_2} \text{ (T-App)}$$

where $t = t_1(t_2)$ and $T = T_2$.

- $t_1 \rightarrow t'_1$ and $t \rightarrow t' = t'_1(t_2)$. By induction hypothesis on \mathcal{D}_1 , we have $\mathcal{D}'_1 :: \Gamma \vdash t'_1 : T_1 \rightarrow T_2$. Hence, $\Gamma \vdash t' : T$ can be derived as follows.

$$\frac{\mathcal{D}'_1 :: \Gamma \vdash t'_1 : T_1 \rightarrow T_2 \quad \mathcal{D}_2 :: \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1(t_2) : T_2} \text{ (T-App)}$$

- t_1 is a value and $t_2 \rightarrow t'_2$ and $t \rightarrow t' = t_1(t'_2)$. By induction hypothesis on \mathcal{D}_2 , we have $\mathcal{D}'_2 :: \Gamma \vdash t'_2 : T_2$. Hence, $\Gamma \vdash t' : T$ can be derived as follows.

$$\frac{\mathcal{D}_1 :: \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \mathcal{D}'_2 :: \Gamma \vdash t'_2 : T_2}{\Gamma \vdash t_1(t_2) : T_2} \text{ (T-App)}$$

- $t_1 = \lambda x : T_1. t'_1$ for some t'_1 and t_2 is a value and $t \rightarrow t' = [x \mapsto t_2]t'_1$. Then \mathcal{D}_1 is of the following form

$$\frac{\Gamma, x : T_1 \vdash t'_1 : T_2}{\Gamma \vdash \lambda x : T_1. t'_1 : T_1 \rightarrow T_2} \text{ (T-Abs)}$$

By the Substitution Lemma, $\Gamma \vdash [x \mapsto t_2]t'_1 : T_2$ is derivable. Note that $t' = [x \mapsto t_2]t'_1$ and we are done. ■

1.5 Evaluation Contexts

evaluation contexts $E ::= [] \mid \text{if } E \text{ then } t_1 \text{ else } t_2 \mid op(E) \mid E(t) \mid v(E)$

Definition 6 A redex is a special form of term.

- **if true then t_1 else t_2** is a redex and its reduction is t_1 .
- **if false then t_1 else t_2** is a redex and its reduction is t_2 .
- $op(v_1)$ is a redex if $op(v_1) = v_2$ and its reduction is v_2 .
- $(\lambda x : T.t)(v)$ is a redex and its reduction is $[x \mapsto v]t$.

Assume that $t_1 = E[t]$ and $t_2 = E[t']$, where t is a redex and t' is its reduction. Then we write $t_1 \rightarrow t_2$ and say that t_1 reduces to t_2 in one step. Let \rightarrow^* be the reflexive and transitive closure of \rightarrow .

Proposition 7 Assume $E[t] = E'[t']$, where t and t' are redexes. Then $E = E'$ and $t = t'$.

Proof By structural induction on E . ■

Theorem 8 (Subject Reduction) Assume $\mathcal{D} :: \emptyset \vdash t : T$ and $t \rightarrow t'$. Then $\emptyset \vdash t' : T$ is derivable.

Proof Assume that $t = E[t_1]$ and $t' = E[t'_1]$, where t_1 is a redex and t'_1 is the reduction of t_1 . The proof proceeds by structural induction on E . ■

Lemma 9 Assume $\mathcal{D} :: \emptyset \vdash t : T$. Then t is a value or $t = E[t_0]$ for some evaluation context E and redex t_0 .

Proof By structural induction on \mathcal{D} . ■

Theorem 10 (Progress) Assume $\mathcal{D} :: \emptyset \vdash t : T$. Then t is a value or $t \rightarrow t'$ for some term t' .

Proof Assume that t is not a value. By Lemma 9, $t = E[t_0]$ for some evaluation context E and redex t_0 . Hence, we have $t \rightarrow t'$ for $t' = E[t'_0]$, where t'_0 is the reduction of the redex t_0 . ■

1.6 Type Erasue and Typability

Chapter 2

Simple Extensions

2.1 Base Types

2.2 The Unit Type

2.3 Derived Forms: Sequencing and Wildcards

terms $t ::= \dots \mid t_1; t_2 \mid \lambda_ : Unit.t$

$$\frac{t_1 \rightarrow t'_1}{t_1; t_2 \rightarrow t'_1; t_2} \text{ (E-Seq)}$$

$$\frac{}{unit; t_2 \rightarrow t_2} \text{ (E-SeqNext)}$$

2.4 Ascription

terms $t ::= \dots \mid t \text{ as } T$

$$\frac{t \rightarrow t'}{t \text{ as } T \rightarrow t' \text{ as } T} \text{ (E-Ascribe)}$$

$$\frac{}{v \text{ as } T \rightarrow v} \text{ (E-AscribeVal)}$$

2.5 Let Bindings

terms $t ::= \dots \mid \text{let } x = t_1 \text{ in } t_2$

$$\frac{t_1 \rightarrow t'_1}{\text{let } x = t_1 \text{ in } t_2 \rightarrow \text{let } x = t'_1 \text{ in } t_2} \text{ (E-Let)}$$

$$\frac{}{\text{let } x = v_1 \text{ in } t_2 \rightarrow [x \mapsto v_1]t_2} \text{ (E-LetVal)}$$

2.6 Pairs

types $T ::= \dots \mid T_1 * T_2$
 terms $t ::= \dots \mid \{t_1, t_2\} \mid t.1 \mid t.2$
 values $v ::= \dots \mid \{v_1, v_2\}$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 * T_2} \text{ (T-Pair)}$$

$$\frac{\Gamma \vdash t : T_1 * T_2}{\Gamma \vdash t.1 : T_1} \text{ (T-Proj1)}$$

$$\frac{\Gamma \vdash t : T_1 * T_2}{\Gamma \vdash t.1 : T_2} \text{ (T-Proj2)}$$

$$\frac{t_1 \rightarrow t'_1}{\{t_1, t_2\} \rightarrow \{t'_1, t_2\}} \text{ (E-Pair1)}$$

$$\frac{t_2 \rightarrow t'_2}{\{v_1, t_2\} \rightarrow \{v_1, t'_2\}} \text{ (E-Pair2)}$$

$$\frac{t \rightarrow t'}{t.1 \rightarrow t'.1} \text{ (E-Proj1)}$$

$$\frac{}{\{v_1, v_2\}.1 \rightarrow v_1} \text{ (E-Proj1Val)}$$

$$\frac{t \rightarrow t'}{t.2 \rightarrow t'.2} \text{ (E-Proj2)}$$

$$\frac{}{\{v_1, v_2\}.2 \rightarrow v_2} \text{ (E-Proj2Val)}$$

2.7 Tuples

2.8 Records

types $T ::= \dots \mid \{l_1 : T_1, \dots, l_n : T_n\}$
 terms $t ::= \dots \mid \{l_1 = t_1, \dots, l_n = t_n\} \mid t.l$

$$\frac{\Gamma t_i : T_i \text{ for } i = 1, \dots, n}{\Gamma \vdash \{l_1 = t_1, \dots, l_n = t_n\} : \{l_1 : T_1, \dots, l_n : T_n\}} \text{ T-Record}$$

2.9 Sums

terms $t ::= \dots \mid \text{inl}(t) \mid \text{inr}(t) \mid \text{case } t \text{ of } \text{inl}(x) \Rightarrow t_1 \mid \text{inr}(x) \Rightarrow t_2$

$$\begin{array}{c}
\frac{\Gamma \vdash \mathit{inl}(t_1) : T_1 + T_2}{\Gamma \vdash t_1 : T_1} \text{ (T-Inl)} \\
\frac{\Gamma \vdash \mathit{inr}(t_2) : T_1 + T_2}{\Gamma \vdash t_2 : T_2} \text{ (T-Inr)} \\
\frac{\Gamma \vdash t : T_1 + T_2 \quad \Gamma, x : T_1 \vdash t_1 : T \quad \Gamma, x : T_2 \vdash t_2 : T}{\Gamma \vdash \mathbf{case } t \mathbf{ of } \mathit{inl}(x) \Rightarrow t_1 \mid \mathit{inr}(x) \Rightarrow t_2 : T} \text{ (T-Case)} \\
\frac{}{\mathbf{case } \mathit{inl}(v) \mathbf{ of } \mathit{inl}(x) \Rightarrow t_1 \mid \mathit{inr}(x) \Rightarrow t_2 \rightarrow [x \mapsto v]t_1} \text{ (E-CaseInl)} \\
\frac{}{\mathbf{case } \mathit{inr}(v) \mathbf{ of } \mathit{inl}(x) \Rightarrow t_1 \mid \mathit{inr}(x) \Rightarrow t_2 \rightarrow [x \mapsto v]t_2} \text{ (E-CaseInr)} \\
\frac{t \rightarrow t'}{\mathbf{case } t \mathbf{ of } \mathit{inl}(x) \Rightarrow t_1 \mid \mathit{inr}(x) \Rightarrow t_2 \rightarrow \mathbf{case } t' \mathbf{ of } \mathit{inl}(x) \Rightarrow t_1 \mid \mathit{inr}(x) \Rightarrow t_2} \text{ (E-Case)}
\end{array}$$

2.10 Variants

2.11 General Recursion

terms $t ::= \dots \mid \mathit{fix}(t) \mid \mathbf{letrec } x : T_1 = t_1 \mathbf{ in } t_2$

$$\begin{array}{c}
\frac{\Gamma \vdash t : T \rightarrow T}{\Gamma \vdash \mathit{fix}(t) : T} \text{ (T-Fix)} \\
\frac{t \rightarrow t'}{\mathit{fix}(t) \rightarrow \mathit{fix}(t')} \text{ (E-Fix)} \\
\frac{}{\mathit{fix}(\lambda x : T.t) \rightarrow [x \mapsto \mathit{fix}(\lambda x : T.t)]t} \text{ (E-FixVal)}
\end{array}$$

Chapter 3

Normalization

Definition 11 Given a term t , let $l(t)$ be the least upper bound on the numbers of reduction steps in reduction sequences starting from t .

By the König's lemma, $l(t) = \infty$ if and only if there is an infinite reduction sequence starting from t . We write $t \downarrow$ to mean $l(t) < \infty$.

Definition 12 (Reducibility) For each type T , we define a predicate R_T on closed terms of type T .

- T is a base type. Then $R_T(t)$ holds if $t \downarrow$.
- $T = T_1 * T_2$. Then $R_T(t)$ holds if $t \downarrow$ and $R_{T_1}(v_1)$ and $R_{T_2}(v_2)$ hold whenever $t \rightarrow^* \{v_1, v_2\}$.
- $T = T_1 \rightarrow T_2$. Then $R_T(t)$ holds if $t \downarrow$ and $R_{T_2}([x \mapsto v_1]t_2)$ holds whenever $t \rightarrow^* \lambda x : T_1.t_2$ and $R_{T_1}(v_1)$.

Proposition 13 Assume that t is a closed expression of type T .

1. If $R_T(t)$, then $t \downarrow$.
2. If $R_T(t)$ and $t \rightarrow t'$, then $R_T(t')$.
3. If t is not a value and $R_T(t')$ for every t' satisfying $t \rightarrow t'$, then $R_T(t)$.

Proof (1) and (2) are straightforward. We now prove (3) by structural induction on T .

- T is a base type. Then $l(t) \leq \max(\{0\} \cup \{1 + l(t') \mid t \rightarrow t'\})$. Therefore, $t \downarrow$, which implies $R_T(t)$.
- $T = T_1 * T_2$ for some types T_1 and T_2 . Assume that $t \rightarrow^* \{v_1, v_2\}$ for some values v_1 and v_2 . Since t is not a value, there exists a term t' of type T such that $t \rightarrow t' \rightarrow^* \{v_1, v_2\}$ holds. Since $R_T(t')$, $R_{T_i}(v_i)$ for $i = 1, 2$. Hence, $R_T(t)$ by the definition of R_T .
- $T = T_1 \rightarrow T_2$ for some types T_1 and T_2 . Assume that $t \rightarrow^* \lambda x : T_1.t_2$. Since t is not a value, there exists a term t' of type T such that $t \rightarrow t'$ and $t' \rightarrow^* \lambda x : T_1.t_2$. Since $R_T(t')$, $R_{T_2}([x \mapsto v_1]t_2)$ for every v_1 satisfying $R_{T_1}(v_1)$. Hence, $R_T(t)$ by the definition of R_T .

Therefore, (3) is valid. ■

Lemma 14 (*Main lemma*) Assume that $\Gamma \vdash t : T$ is derivable and $\vdash \theta : \Gamma$ holds. If $R_{\Gamma(x)}(\theta(x))$ for all $x \in \mathbf{dom}(\theta) = \mathbf{dom}(\Gamma)$, then $R_T(t[\theta])$.

Proof By structural induction on a derivation \mathcal{D} of $\Gamma \vdash t : T$.

- \mathcal{D} is of the following form,

$$\frac{\mathcal{D}_0 :: \Gamma \vdash t_0 : \mathit{Bool} \quad \mathcal{D}_1 :: \Gamma \vdash t_1 : T \quad \mathcal{D}_2 \Gamma \vdash t_2 : T}{\Gamma \vdash \mathbf{if} \ t_0 \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2 : T} \quad (\mathbf{T-If})$$

where $t = \mathbf{if} \ t_0 \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2$. By induction hypothesis on \mathcal{D}_0 , $R_{\mathit{Bool}}(t_0[\theta])$ holds. Also, $R_T(t_1[\theta])$ and $R_T(t_2[\theta])$ hold by induction hypothesis on \mathcal{D}_1 and \mathcal{D}_2 , respectively.

We now prove by induction on $l(t_0)$ that if $R_{\mathit{Bool}}(t_0), R_T(t_1)$ and $R_T(t_2)$, then $R_T(\mathbf{if} \ t_0 \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2)$. Assume that $t = R_T(\mathbf{if} \ t_0 \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2) \rightarrow t'$.

- $t_0 \rightarrow t'_0$ and $t \rightarrow t' = \mathbf{if} \ t'_0 \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2$. Then $l(t'_0) < l(t_0)$ and by induction hypothesis on t'_0 , $R_T(t')$ holds.
- $t_0 = \mathbf{true}$ and $t \rightarrow t' = t_1$. Then obviously $R_T(t')$ holds.
- $t_0 = \mathbf{false}$ and $t \rightarrow t' = t_2$. Then obviously $R_T(t')$ holds.

Hence by Proposition 13 (3), $R_T(t)$ holds.

We can now conclude that $R_T(t[\theta])$ holds as $t[\theta] = \mathbf{if} \ t_0[\theta] \ \mathbf{then} \ t_1[\theta] \ \mathbf{else} \ t_2[\theta]$.

- \mathcal{D} is of the following form,

$$\frac{\mathcal{D}_1 :: \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \mathcal{D}_2 :: \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1(t_2) : T_2} \quad (\mathbf{T-App})$$

where $t = t_1(t_2)$ and $T = T_2$. Assume $\theta : \Gamma$. By induction hypothesis on \mathcal{D}_1 and \mathcal{D}_2 , both $R_{T_1 \rightarrow T_2}(t_1[\theta])$ and $R_{T_2}(t_2[\theta])$ hold.

We now prove by induction on $l(t_1) + l(t_2)$ that for all terms t_1 and t_2 , $R_{T_1 \rightarrow T_2}(t_1)$ and $R_{T_2}(t_2)$ implies $R_{T_2}(t_1(t_2))$. Assume that $t \rightarrow t'$. There are three possibilities.

- $t_1 \rightarrow t'_1$ and $t' = t'_1(t_2)$. Since $l(t'_1) + l(t_2) < l(t_1) + l(t_2)$, $R_{T_2}(t')$ holds by induction hypothesis.
- $t_1 = v_1$ for some value v_1 , $t_2 \rightarrow t'_2$ and $t' = t_1(t'_2)$. Since $l(t_1) + l(t'_2) < l(t_1) + l(t_2)$, $R_{T_2}(t')$ holds by induction hypothesis.
- $t_1 = \lambda x : T_1.t_{11}$ for some term t_{11} , $t_2 = v_2$ for some value v_2 and $t' = [x \mapsto v_2]t_{11}$. Since $R_{T_1 \rightarrow T_2}(t_1)$ and $R_{T_2}(t_2)$, $R_{T_2}(t')$ by the definition of $R_{T_1 \rightarrow T_2}$.

Therefore, $R_T(t)$ by Proposition 13 (3).

We can now conclude that $R_{T_2}(t[\theta])$ holds as $t[\theta] = t_1[\theta](t_2[\theta])$.

- \mathcal{D} is of the following form,

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\mathcal{D}_1 :: \Gamma \vdash \lambda x : T_1.t_2 : T_1 \rightarrow T_2} \quad (\mathbf{T-Abs})$$

where $t = \lambda x : T_1.t_2$ and $T = T_1 \rightarrow T_2$.

Assume $\theta : \Gamma$. Let $t^* = t[\theta] = \lambda x : T_1.t_2[\theta]$ and we need to show $R_T(t^*)$. Assume $R_{T_1}(v_1)$ for some value v_1 . Then $[x \mapsto v_1]t_2[\theta] = t_2[\theta[x \mapsto v_1]]$. Note that $\theta[x \mapsto v_1] : \Gamma, x : T_1$ holds. By induction hypothesis on \mathcal{D}_1 , $R_{T_2}(t_2[\theta[x \mapsto v_1]])$ holds. Therefore, $R_T(t^*)$ by the definition of R_T .

The rest of cases are trivial. ■

Theorem 15 *If t is a well-typed closed term, then $t \downarrow$.*

Proof By Lemma 14, $R_T(t)$. Therefore, $t \downarrow$ by Proposition 13 (1). ■

Chapter 4

References

4.1 Introduction

terms $t ::= \dots \mid \text{ref}(t) \mid !t \mid t_1 := t_2 \mid l$
values $v ::= \dots \mid l$
types $T ::= \dots \mid \text{Ref}(T)$

Note that we use l for a reference, which is a value.

Here is an example where recursion is achieved through reference.

```
val fact: (int -> int) ref = ref (fn n => 0)

val _ = fact := (fn (n: int): int => if n = 0 then 1 else n * !fact (n-1))
```

4.2 Typing

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \text{ref}(t) : \text{Ref}(T)} \text{ (T-Ref)}$$
$$\frac{\Gamma \vdash t : \text{Ref}(T)}{\Gamma \vdash !t : T} \text{ (T-DeRef)}$$
$$\frac{\Gamma \vdash t_1 : \text{Ref}(T) \quad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 := t_2 : \text{Unit}} \text{ (T-Assign)}$$

4.3 Evaluation

The existing evaluation rules can be augmented with a store straightforwardly. For instance, we now have the following evaluation rules.

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{t_1(t_2) \mid \mu \rightarrow t'_1(t_2) \mid \mu'} \text{ (E-App1)}$$

$$\frac{t_2 \mid \mu \rightarrow t'_2 \mid \mu'}{v_1(t_2) \mid \mu \rightarrow v_1(t'_2) \mid \mu'} \text{ (E-App2)}$$

$$\frac{}{(\lambda x.t_1)(v_2) \mid \mu \mapsto [x \mapsto v_2]t_1 \mid \mu} \text{ (E-AppAbs)}$$

Here are the new evaluation rules.

$$\frac{t \mid \mu \rightarrow t' \mid \mu'}{\text{ref}(t) \mid \mu \rightarrow \text{ref}(t') \mid \mu'} \text{ (E-Ref)}$$

$$\frac{l \notin \mathbf{dom}(\mu)}{\text{ref}(v) \mid \mu \rightarrow l \mid (\mu, l \mapsto v)} \text{ (E-RefVal)}$$

$$\frac{!t \mid \mu \rightarrow !t' \mid \mu'}{t \mid \mu \rightarrow t' \mid \mu'} \text{ (E-DeRef)}$$

$$\frac{l \in \mathbf{dom}(\mu)}{!l \mid \mu \rightarrow \mu(l) \mid \mu} \text{ (E-DeRefVal)}$$

$$\frac{l \in \mathbf{dom}(\mu)}{l := v \mid \mu \rightarrow \text{unit} \mid \mu[l := v]} \text{ (E-AssignVal)}$$

4.4 Store Typings

$$\frac{\Sigma(l) = T}{\Gamma \mid \Sigma \vdash l : \text{Ref}(T)} \text{ (T-Loc)}$$

$$\frac{\Gamma \mid \Sigma \vdash t : T}{\Gamma \mid \Sigma \vdash \text{ref}(t) : \text{Ref}(T)} \text{ (T-Ref)}$$

$$\frac{\Gamma \mid \Sigma \vdash !t : T}{\Gamma \mid \Sigma \vdash t : \text{Ref}(T)} \text{ (T-DeRef)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref}(T) \quad \Gamma \vdash t_2 : T}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}} \text{ (T-Assign)}$$

4.5 Safety

Definition 16 A store μ is said to be well-typed with respect to a typing context Γ and a store typing Σ , written $\Gamma \mid \Sigma \vdash \mu$ if $\mathbf{dom}(\mu) = \mathbf{dom}(\Sigma)$ and $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in \mathbf{dom}(\mu) = \mathbf{dom}(\Sigma)$.

Theorem 17 (Type Preservation) If $\Gamma \mid \Sigma \vdash t : T$, $\Gamma \mid \Sigma \vdash \mu$ and $t \mid \mu \rightarrow t' \mid \mu'$, then for some $\Sigma' \supseteq \Sigma$, $\Gamma \mid \Sigma' \vdash t' : T'$ and $\Gamma \mid \Sigma' \vdash \mu'$.

Theorem 18 (*Progress*) *Suppose $\mathcal{D} :: \emptyset \mid \Sigma \vdash t : T$. Then either t is a value or for any store μ such that $\emptyset \mid \Sigma \vdash \mu$, $t \mid \mu \rightarrow t' \mid \mu'$ holds for some t' and μ' .*

Chapter 5

Exceptions

Exceptions provide a means to altering the normal control in the execution of a program. For instance, the following example indicates a way to terminate the execution of a loop by executing the `break` command.

```
while (test) {  
  ...  
  if (something_is_true) break;  
  ...  
}
```

This can be readily implemented through the use of exceptions. A raised exception may be captured later so as to be handled by an exception handler.

```
datatype binaryTree = E | B of binaryTree * binaryTree  
  
fun isBraunTree (t: binaryTree): bool =  
  let  
    exception NotBraunTree  
  
    fun aux (E) = 0  
      | aux (B (t1, t2)) =  
        let  
          val ls = aux t1 and rs = aux t2  
        in  
          if ls = rs orelse ls = rs + 1 then 1 + ls + rs  
          else raise NotBraunTree  
        end  
  in  
    let  
      val _ = aux t  
    in  
      true  
    end handle NotBraunTree => false  
  end
```

5.1 Raising and Handling Exceptions

constants $c ::= \dots \mid \text{error}$
 terms $t ::= \dots \mid \text{raise}(t) \mid \text{try } t_1 \text{ with } t_2$
 answers $ans ::= v \mid \text{raise}(v)$

$$\begin{array}{c}
 \frac{\Gamma \vdash t : \text{Exn}}{\Gamma \vdash \text{raise}(t) : T} \text{ (T-Raise)} \\
 \\
 \frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : \text{Exn} \rightarrow T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T} \text{ (T-TryWith)} \\
 \\
 \frac{t \rightarrow t'}{\text{raise}(t) \rightarrow \text{raise}(t')} \text{ (E-Raise)} \\
 \\
 \frac{}{\text{raise}(v)(t_2) \rightarrow \text{raise}(v)} \text{ (E-Raise-App1)} \\
 \\
 \frac{}{v_1(\text{raise}(v)) \rightarrow \text{raise}(v)} \text{ (E-Raise-App2)} \\
 \\
 \frac{t_1 \rightarrow t'_1}{\text{try } t_1 \text{ with } t_2 \rightarrow \text{try } t'_1 \text{ with } t_2} \text{ (E-TryWith)} \\
 \\
 \frac{}{\text{try } v_1 \text{ with } t_2 \rightarrow v_1} \text{ (E-TryWithVal)} \\
 \\
 \frac{}{\text{try } \text{raise}(v) \text{ with } t_2 \rightarrow t_2(v)} \text{ (E-TryWithExn)}
 \end{array}$$

Theorem 19 (*Subject Reduction*) Assume that $\mathcal{D} :: \Gamma \vdash t : T$ and $t \rightarrow t'$. Then $\Gamma \vdash t' : T$ is derivable.

Theorem 20 (*Progress*) Assume that $\mathcal{D} :: \emptyset \vdash t : T$. Then either t is an answer or $t \rightarrow t'$ for some term t' .

Chapter 6

Recursive Types

6.1 Examples

$$IntList = \langle nil : Unit, cons : Int * IntList \rangle$$

$$IntList = \mu X. \langle nil : Unit, cons : Int * X \rangle$$

$$\begin{aligned} nil &= \langle nil = unit \rangle \\ cons &= \lambda x : Int. \lambda xs : tIntList. \langle cons = \{x, xs\} \rangle \\ isNil &= \lambda xs. \mathbf{case} \ xs \ \mathbf{of} \ \langle nil = u \rangle \Rightarrow \mathbf{true} \mid \langle cons = p \rangle \Rightarrow \mathbf{false} \\ hd &= \lambda xs. \mathbf{case} \ xs \ \mathbf{of} \ \langle nil = u \rangle \Rightarrow \mathbf{raise}(EmptyList) \mid \langle cons = p \rangle \Rightarrow p.1 \\ tl &= \lambda xs. \mathbf{case} \ xs \ \mathbf{of} \ \langle nil = u \rangle \Rightarrow \mathbf{raise}(EmptyList) \mid \langle cons = p \rangle \Rightarrow p.2 \end{aligned}$$

$$T_1 = \mu X. Int \rightarrow X$$

$$T_2 = \mu X. X \rightarrow Int$$

$$f_1 = \mathbf{fix}(\lambda f : Int \rightarrow T_1. \lambda n : Int. f)$$

$$\Omega = (\lambda x : T_2. x(x))(\lambda x : T_2. x(x))$$

$$Stream = \mu X. Unit \rightarrow Int * X$$

$$Process = \mu X. Int \rightarrow Int * X$$

$$upFrom = \mathbf{fix}(\lambda f : Int \rightarrow Stream. \lambda n : Int. \lambda _ : Unit. n * f(n + 1))$$

$$Nats = upFrom(0)$$

In the presence of recursive types, we can construct a nonterminating function without using the fixed point operator *fix*.

```
datatype T = C of T -> T
fun f (x: T): T = case x of C g => g (x)
```

Note that the evaluation of $f(C(f))$ is nonterminating.

Here is a representation of untyped λ -terms through the use of higher-order abstract syntax.

datatype T = Lam of T -> T | App of T * T

$$\begin{aligned} |x| &= x \\ |\lambda x.t| &= Lam(\lambda x : T. |t|) \\ |t_1(t_2)| &= App(|t_1|, |t_2|) \end{aligned}$$

6.2 Formalization

types $T ::= \dots | X | \mu X.T$

$$\begin{aligned} fold([X \mapsto \mu X.T]T) &= \mu X.T \\ unfold(\mu X.T) &= [X \mapsto \mu X.T]T \end{aligned}$$

$$\frac{U = \mu X.T \quad \Gamma \vdash t : [X \mapsto U]T}{\Gamma \vdash t : U} \text{ (T-Fold)}$$

$$\frac{U = \mu X.T \quad \Gamma \vdash t : U}{\Gamma \vdash t : [X \mapsto U]T} \text{ (T-Unfold)}$$

Chapter 7

Universal Types

7.1 System F

types	$T ::= X \mid T \rightarrow T \mid \forall X.T$
terms	$t ::= x \mid \lambda x : T.t \mid t_1(t_2) \mid \Lambda X.t \mid t[T]$
values	$v ::= \lambda x : T.t \mid \Lambda X.t$
contexts	$\Gamma ::= \emptyset \mid \Gamma, x : T$

Encoding Datatypes

Pairs Let us define $Pair(T_1, T_2)$ as $\forall X.(T_1 \rightarrow T_2 \rightarrow X) \rightarrow X$.

$$\begin{aligned} pair(x, y) &= \Lambda X.\lambda p : T_1 \rightarrow T_2 \rightarrow X.p(x)(y) \\ fst(p) &= p[T_1](\lambda x : T_1 \lambda y : T_2.x) \\ snd(p) &= p[T_2](\lambda x : T_1 \lambda y : T_2.y) \end{aligned}$$

Natural Numbers Let us define Nat as $\forall X.X \rightarrow (X \rightarrow X) \rightarrow X$. Then the two constructors Z and S can be defined as follows.

$$\begin{aligned} Z &= \Lambda X.\lambda z : X.\lambda s : X \rightarrow X.z \\ S(n : Nat) &= \Lambda X.\lambda z : X.\lambda s : X \rightarrow X.s(n[X](z)(s)) \end{aligned}$$

Lists Let us define $List(T)$ as $\forall X.X \rightarrow (T \rightarrow X \rightarrow X) \rightarrow X$. Then the two list constructors Nil and $Cons$ can be defined as follows.

$$\begin{aligned} Nil &= \Lambda X.\lambda nil : X.\lambda cons : T \rightarrow X \rightarrow X.nil \\ Cons(x : T)(xs : List(T)) &= \Lambda X.\lambda nil : X.\lambda cons : T \rightarrow X \rightarrow X.cons(x)(xs[X](nil)(cons)) \end{aligned}$$

7.2 Some Basic Properties

Definition 21 Given a type variable X , a type T and a context Γ , we write $[X \mapsto T]\Gamma$ for the context Γ' such that $\mathbf{dom}(\Gamma) = \mathbf{dom}(\Gamma')$ and $\Gamma'(x) = [X \mapsto T](\Gamma(x))$.

Lemma 22 Assume $\mathcal{D} :: \vec{X}; \Gamma \vdash t : T$ and $\vec{X} \vdash T_1$ [type]. Then $\vec{X}; [X \mapsto T_1]\Gamma \vdash [X \mapsto T_1]t : [X \mapsto T_1]T$ is derivable.

$$\begin{array}{c}
\overline{\vec{X} \vdash \emptyset [ctx]} \\
\frac{\vec{X} \vdash \Gamma [ctx] \quad \vec{X} \vdash T [ctx]}{\vec{X} \vdash \Gamma, x : T [ctx]} \\
\frac{X \text{ is in } \vec{X}}{\vec{X} \vdash X [type]} \text{ (K-Var)} \\
\frac{\vec{X} \vdash T_1 [type] \quad \vec{X} \vdash T_2 [type]}{\vec{X} \vdash T_1 \rightarrow T_2 [type]} \text{ (K-Fun)} \\
\frac{\vec{X}, X \vdash T [type]}{\vec{X} \vdash \forall X.T [type]} \text{ (K-Uni)} \\
\frac{\vec{X} \vdash \Gamma [ctx] \quad \Gamma(x) = T}{\vec{X}; \Gamma \vdash x : T} \text{ (T-Var)} \\
\frac{\vec{X}; \Gamma, x : T_1 \vdash t : T_2}{\vec{X}; \Gamma \vdash \lambda x : T_1. t : T_1 \rightarrow T_2} \text{ (T-Abs)} \\
\frac{\vec{X}; \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \vec{X}, \Gamma \vdash t_2 : T_1}{\vec{X}; \Gamma \vdash t_1(t_2) : T_2} \text{ (T-App)} \\
\frac{\vec{X}, X; \Gamma \vdash t : T \quad \vec{X} \vdash \Gamma [ctx]}{\vec{X}; \Gamma \vdash \Lambda X.t : \forall X.T} \text{ (T-Tabs)} \\
\frac{\vec{X}; \Gamma \vdash t : \forall X.T \quad \vec{X} \vdash T_0 [type]}{\vec{X}; \Gamma \vdash t[T_0] : [X \mapsto T_0]T} \text{ (T-Tapp)} \\
\frac{t_1 \rightarrow t'_1}{t_1(t_2) \rightarrow t'_1(t_2)} \text{ (E-App1)} \\
\frac{t_2 \rightarrow t'_2}{v_1(t_2) \rightarrow v_1(t'_2)} \text{ (E-App2)} \\
\frac{t_1 \rightarrow t'_1}{t_1(t_2) \rightarrow t'_1(t_2)} \text{ (E-App2)} \\
\overline{(\lambda x : T_1.t_1)(v_2) \rightarrow [x \mapsto v_2]t_1} \text{ (E-AppAbs)} \\
\frac{t[T] \rightarrow t'[T]}{t \rightarrow t'} \text{ (E-Tapp)} \\
\overline{(\Lambda X.t)[T] \rightarrow [X \mapsto T]t} \text{ (E-TappTabs)}
\end{array}$$

Proof The proof follows from structural induction on \mathcal{D} . ■

Theorem 23 (*Subject Reduction*) Assume that $\mathcal{D} :: \vec{X}; \Gamma \vdash t : T$ and $t \rightarrow t'$. Then $\vec{X}; \Gamma \vdash t' : T$ is derivable.

Proof We proceed by structural induction on \mathcal{D} .

- \mathcal{D} has the following form,

$$\frac{\mathcal{D}_1 :: \vec{X}; \Gamma \vdash t_1 : \forall X.T_1 \quad \vec{X} \vdash T_2 \text{ [type]}}{\vec{X}; \Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_1} \text{ (T-Tapp)}$$

where $t = t_1[T_2]$ and $T = [X \mapsto T_2]T_1$. We now have two subcases.

- $t_1 \rightarrow t'_1$ and $t \rightarrow t' = t'_1[T_2]$. By induction hypothesis on \mathcal{D}_1 , we have $\mathcal{D}'_1 :: \Gamma \vdash t'_1 : \forall X.T_1$ for some \mathcal{D}'_1 , and therefore $\vec{X}; \Gamma \vdash t' : T$ is derivable.
- $t_1 = \Lambda X.t'_1$ and $t \rightarrow t' = [X \mapsto T_2]t'_1$. Then \mathcal{D}_1 has the following form:

$$\frac{\vec{X}, X; \Gamma \vdash t_1 : T_1 \quad \vec{X} \vdash \Gamma \text{ [ctx]}}{\vec{X}; \Gamma \vdash \Lambda X.t'_1 : \forall X.T_1} \text{ (T-Tabs)}$$

where $t = \Lambda X.t_1$ and $T = \forall X.T_1$. By Lemma 22, $\vec{X}; \Gamma \vdash [X \mapsto T_2]t'_1 : [X \mapsto T_2]T_1$ is derivable since Γ contains no free occurrences of X . Note that $t' = [X \mapsto T_2]t'_1$ and $T = [X \mapsto T_2]T_1$, and we are done.

The other cases can be handled similarly. ■

Theorem 24 (*Progress*) Assume $\mathcal{D} :: \vec{X}; \Gamma \vdash t : T$. Then t is a value or $t \rightarrow t'$ holds for some term t' .

Proof The proof follows from structural induction on \mathcal{D} . ■

7.3 Type Erasure

We can define a type erasure function $|\cdot|$ as follows, which translates a term in System F into an untyped λ -term.

$$|x| = x \quad |\lambda x : T.t| = \lambda x.t \quad |t_1(t_2)| = |t_1|(|t_2|) \quad |\Lambda X.t| = |t| \quad |t[T]| = |t|$$

Notice that for a value v in System F, $|v|$ may not necessarily be a value. Therefore, given a term t in System F, $|t| \rightarrow^* v_0$ does not necessarily imply that we have $t \rightarrow^* v$ for some value v such that $|v| = v_0$. To have this property, we can impose a restriction on **(T-Tabs)** by requiring that t be a value whenever the following rule is applied.

$$\frac{\vec{X}, X; \Gamma \vdash t : T \quad \vec{X} \vdash \Gamma \text{ [ctx]}}{\vec{X}; \Gamma \vdash \Lambda X.t : \forall X.T} \text{ (T-Tabs)}$$

This restriction is often called *value restriction*.

7.4 Normalization for System F

Definition 25 Given a closed type T , a predicate R defined on the closed terms of the type T is a reducibility candidate of type T if R satisfies the following conditions.

- If $R(t)$ then $t \downarrow$.
- If $R(t)$ and $t \rightarrow t'$ then $R(t')$.
- If t is not a value and $R(t')$ whenever $t \rightarrow t'$, then $R(t)$.

For instance, the predicate $R(t) \equiv t \downarrow$ defined on the closed terms of a given type T is a reducibility candidate of the type T .

We use Θ_{typ} for a substitution that maps type variables to types and Θ_{red} for a finite map that maps type variables to reducibility candidates. We write $\Theta_{red} : \Theta_{typ}$ if for each $X \in \mathbf{dom}(\Theta_{red}) = \mathbf{dom}(\Theta_{typ})$, $\Theta_{red}(X)$ is a reducibility candidate of type $\Theta_*(X)$. In addition, we write $\Theta_{typ} : \vec{X}$ if $\mathbf{dom}(\Theta_{typ}) = \{\vec{X}\}$.

Definition 26 Assume that $\vec{X} \vdash T$ [type] is derivable and $\Theta_{red} : \Theta_{typ} : \vec{X}$. We define $Red[\Theta_{red}; \Theta_{typ}; T]$ as follows by structural induction on T .

- T is a type variable. Then $Red[\Theta_{red}; \Theta_{typ}; T] = \Theta_{red}(T)$.
- $T = T_1 \rightarrow T_2$. Then $Red[\Theta_{red}; \Theta_{typ}; T](t)$ if $t \downarrow$ and whenever $t \rightarrow^* \lambda x : T_1[\Theta_{typ}].t_2$, we have $Red[\Theta_{red}; \Theta_{typ}; T_2](\llbracket x \mapsto v_1 \rrbracket t_2)$ for each v_1 satisfying $Red[\Theta_{red}; \Theta_{typ}; T_1](v_1)$.
- $T = \forall X.T_1$. Then $Red[\Theta_{red}; \Theta_{typ}; T](t)$ if $t \downarrow$ and whenever $t \rightarrow^* \Lambda X.t_1$, $Red[\Theta_{red}[X \mapsto R]; \Theta_{typ}[X \mapsto T_2]; T_1](\llbracket X \mapsto T_2 \rrbracket t_1)$ holds for each reducibility candidate R of T_2 .

Proposition 27 $Red[\Theta_{red}; \Theta_{typ}; T]$ is a reducibility candidate of the type $T[\Theta_{typ}]$.

Proof The proof immediately follows from structural induction on T . ■

Proposition 28 We have the following:

$$Red[\Theta_{red}; \Theta_{typ}; [X \mapsto T_1]T] = Red[\Theta_{red}[X \mapsto Red[\Theta_{red}; \Theta_{typ}; T_1]]; \Theta_{typ}[X \mapsto T_1[\Theta_{typ}]]; T]$$

Proof The proof immediately follows from structural induction on T . ■

Lemma 29 Assume $\mathcal{D} :: \vec{X}; \Gamma \vdash t : T$ and $\Theta_{red} : \Theta_{typ} : \vec{X}$. Also, assume $\theta : \Gamma[\Theta_{typ}]$ and $Red[\Theta_{red}; \Theta_{typ}; \Gamma(x)](\theta(x))$ for each $x \in \mathbf{dom}(\theta)$. Then $Red[\Theta_{red}; \Theta_{typ}; T](t[\Theta_{typ}][\theta])$ holds.

Proof We proceed by structural induction on \mathcal{D} . ■

Theorem 30 (Normalization) Assume that $\emptyset; \emptyset \vdash t : T$ is derivable. Then $t \downarrow$ holds.

Proof By Lemma 29, $Red[\emptyset; \emptyset; T](t)$ holds. Since $Red[\emptyset; \emptyset; T]$ is a reducibility candidate by Proposition 27, $t \downarrow$ holds. ■

Chapter 8

Existential Types

8.1 Introduction

types $T ::= \dots \mid \exists X.T$
term $t ::= \dots \mid \{*T_1, t\} \text{ as } T_2 \mid \text{open } t_1 \text{ as } \{*X, x\} \text{ in } t_2$
values $v ::= \dots \mid \{*T_1, v\} \text{ as } T_2$

$$\frac{\vec{X} \vdash S \text{ [type]} \quad \vec{X}; \Gamma \vdash t : [X \mapsto S]T}{\vec{X}; \Gamma \vdash \{*S, t\} \text{ as } \exists X.T : \exists X.T} \text{ (T-Pack)}$$

$$\frac{\vec{X}; \Gamma \vdash t_1 : \exists X.T_1 \quad \vec{X}, X; \Gamma, x : T_1 \vdash t_2 : T_2 \quad \vec{X} \vdash T_2 \text{ [type]}}{\vec{X}; \Gamma \vdash \text{open } t_1 \text{ as } \{*X, x\} \text{ in } t_2 : T_2} \text{ (T-Unpack)}$$

$$\frac{t \rightarrow t'}{\{*T_1, t\} \text{ as } T_2 \rightarrow \{*T_1, t'\} \text{ as } T_2} \text{ (E-Pack)}$$

$$\frac{t_1 \rightarrow t'_1}{\text{open } t_1 \text{ as } \{*X, x\} \text{ in } t_2 \rightarrow \text{open } t'_1 \text{ as } \{*X, x\} \text{ in } t_2} \text{ (E-Unpack)}$$

$$\frac{}{\text{open } (\{*T_1, v\} \text{ as } T_2) \text{ as } \{*X, x\} \text{ in } t_2 \rightarrow [x \mapsto v][X \mapsto T_1]t_2} \text{ (E-UnpackVal)}$$

8.2 Data Abstraction with Existentials

$Pair[T_1][T_2] : \exists X.\{pair : T_1 \rightarrow T_2 \rightarrow X, fst : X \rightarrow T_1, snd : X \rightarrow T_2\}$
 $Sum[T_1][T_2] : \exists X.\{inl : T_1 \rightarrow X, inr : T_2 \rightarrow X, caseof : \forall Y.X \rightarrow (T_1 \rightarrow Y) \rightarrow (T_2 \rightarrow Y) \rightarrow Y\}$

8.3 Encoding Existentials

$\exists X.T = \forall Y.(\forall X.T \rightarrow Y) \rightarrow Y$
 $\{*S, t\} \text{ as } \exists X.T = \text{let } x = t \text{ in } \Lambda Y.\lambda f : \forall X.T \rightarrow Y.f[S](x)$
 $\text{open } t_1 \text{ as } \{*X, x\} \text{ in } t_2 = t_1[T_2](\Lambda X.\lambda x : T_1.t_2)$

```

{*(T_1 * T_2),
  {pair = \x:T_1.\y:T_2.{x, y}
   fst = \p:T_1 * T_2. p.1
   snd = \p:T_1 * T_2. p.2}}

{*(T_2 * T_1),
  {pair = \x:T_1.\y:T_2.{y, x}
   fst = \p:T_1 * T_2. p.2
   snd = \p:T_1 * T_2. p.1}}

{*(forall X.(T_1 -> T_2 -> X) -> X),
  {pair = \x:T_1.\y:T_2.\X.\p:T_1 -> T_2 -> X. p(x)(y),
   fst = \p:forall X.(T_1 -> T_2 -> X) -> X.p[T_1](\x:T_1.\y:T_2.x),
   snd = \p:forall X.(T_1 -> T_2 -> X) -> X.p[T_1](\x:T_1.\y:T_2.y)}}

{*(forall X.(T_1 -> X) -> (T_2 -> X) -> X),
  {inl = \x:T_1.\X.\l:T_1 -> X.\r:T_2 -> X.l(x),
   inr = \x:T_1.\X.\l:T_1 -> X.\r:T_2 -> X.r(x),
   caseof =
    \Y.\s:forall X.(T_1 -> X) -> (T_2 -> X) -> X.
     \lc:T_1 -> Y.\rc:T_2 -> Y. s[Y](lc)(rc)
  }}

```

```
signature STACK =
  sig
    type T

    val new: unit -> T
    val isEmpty: T -> bool
    val length: T -> int
    val pop: T -> int * T
    val push: int * T -> T
  end

structure Stack: STACK =
  struct
    type T = int list

    fun new () = []

    val isEmpty = List.isEmpty
    val length = List.length

    fun pop (s: T): int * T =
      case s of [] => raise EmptyStack | x :: xs => (x, xs)

    fun push (e: int, s: T): T = e :: s
  end
```


Chapter 9

Type Reconstruction

It can be burdensome for the programmer to write types when programming. Therefore, there is an immediate question as to whether we can find a satisfactory approach that allows the programmer to omit writing types and then infers the omitted types from the structure of a program. We present a positive answer to this question in this section.

9.1 External Language

We present an external language for TFPL_0 . In this language, the programmer is allowed to omit writing types when defining a function **fun** $f(x)$ **is** e . On the other hand, the programmer is also allowed to write $(e : \tau)$ to indicate that the expression e must have type τ .

$$\begin{aligned} \text{expressions } e ::= & x \mid f \mid b \mid i \mid op(e_1, \dots, e_n) \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid \\ & \{ \} \mid \{e_1, e_2\} \mid e.1 \mid e.2 \mid \\ & \text{fun } f(x) \text{ is } e \mid \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e \mid \\ & \text{app}(e_1, e_2) \mid \text{let } x = e_1 \text{ in } e_2 \mid (e : \tau) \end{aligned}$$

Note that we define the type erasure of $(e : \tau)$ as $|e|$, that is, $|(e : \tau)| = |e|$.

9.2 Type Reconstruction Algorithm

We need the following syntax for presenting a type inference algorithm for TFPL_0 .

Existential Type Variables	X
Types	$T ::= X \mid \text{bool} \mid \text{int} \mid \mathbf{1} \mid T * T \mid T \rightarrow T$
Contexts	$G ::= \cdot \mid G, x : T$
Type Constraints	$\Phi ::= \top \mid T_1 \doteq T_2 \mid \Phi_1 \wedge \Phi_2$
Substitution	$\Theta ::= [] \mid \Theta[X \mapsto \tau]$

Given a substitution Θ , the satisfiability of Φ under Θ is defined as follows.

- Φ is satisfied under Θ if Φ is \top ;
- Φ is satisfied under Θ if Φ is $T_1 \doteq T_2$ and $T_1[\Theta] = T_2[\Theta]$.
- Φ is satisfied under Θ if Φ is $\Phi_1 \wedge \Phi_2$ and both Φ_1 and Φ_2 are satisfied under Θ

We say that a type constraint Φ is satisfiable if Φ is satisfiable under some substitution Θ .

We first introduce two kinds of judgments; a judgment $G \vdash e \uparrow T \Rightarrow \Phi$ basically means that we can generate T and Φ for given G and e such that type constraint Φ needs to be solved for typing e with T under context G ; a judgment $G \vdash e \downarrow T \Rightarrow \Phi$ basically states that we can generate Φ for given G , e and T such that type constraint Φ needs to be solved for typing e with type T under context G . The rules for a bidirectional type inference algorithm are presented in Figure 9.1 and Figure 9.2. The following theorem establishes the soundness of these rules.

Theorem 31 (*Soundness*) *We have the following.*

1. *Assume that $G \vdash e \uparrow T \Rightarrow \Phi$ is derivable. If Φ is satisfiable under Θ and $\mathbf{dom}(\Theta)$ contains all existential variables in the judgment $G \vdash e \uparrow T \Rightarrow \Phi$, then $G[\Theta] \vdash |e|[\Theta] : T[\Theta]$ is derivable.*
2. *Assume that $G \vdash e \downarrow T \Rightarrow \Phi$ is derivable. If Φ is satisfiable under Θ and $\mathbf{dom}(\Theta)$ contains all existential variables in the judgment $G \vdash e \downarrow T \Rightarrow \Phi$, then $G[\Theta] \vdash |e|[\Theta] : T[\Theta]$ is derivable.*

Proof (1) and (2) are proven simultaneously with structural induction on the derivations \mathcal{D} of $G \vdash e \uparrow T \Rightarrow \Phi$ and $G \vdash e \downarrow T \Rightarrow \Phi$. ■

Lemma 32 *Let Θ be a substitution on existential variables. We have the following.*

1. *If $G \vdash e \uparrow T \Rightarrow \Phi$ is derivable, then $G[\Theta] \vdash e \uparrow T[\Theta] \Rightarrow \Phi[\Theta]$ is also derivable.*
2. *If $G \vdash e \downarrow T \Rightarrow \Phi$ is derivable, then $G[\Theta] \vdash e \downarrow T[\Theta] \Rightarrow \Phi[\Theta]$ is also derivable.*

Proof (1) and (2) are proven simultaneously with structural induction on the derivations of $G \vdash e \uparrow T \Rightarrow \Phi$ and $G \vdash e \downarrow T \Rightarrow \Phi$. ■

The following theorem establishes the completeness of the presented type inference rules.

Theorem 33 (*Completeness*) *Assume that $\Gamma \vdash e : \tau$ is derivable. Then we have the following.*

1. *$\Gamma \vdash |e| \uparrow T \Rightarrow \Phi$ is derivable for some T and Φ and there exists Θ such that $T[\Theta] = \tau$ and Φ is satisfiable under Θ .*
2. *$\Gamma \vdash |e| \downarrow \tau \Rightarrow \Phi$ is derivable for some Φ and Φ is satisfiable.*

Proof (1) and (2) are proven simultaneously with structural induction on the derivation of $\Gamma \vdash e : \tau$. ■

9.3 Mutual Recursion

We present both typing rules and type inference rules for handling mutual recursion.

types $\tau ::= \dots \mid (\tau_1, \dots, \tau_n)$
expressions $e ::= \dots \mid \mathbf{fun} f_1(x_1 : \tau_1) : \tau'_1 \mathbf{is} e_1 \mathbf{and} \dots \mathbf{and} \mathbf{fun} f_n(x_n : \tau_n) : \tau'_n \mathbf{is} e_n \mid e \# n$

$$\begin{array}{c}
\frac{}{G \vdash b \uparrow \text{bool} \Rightarrow \top} \text{ (tc-bool-up)} \\
\frac{}{G \vdash b \downarrow T \Rightarrow T \doteq \text{bool}} \text{ (tc-bool-dn)} \\
\frac{}{G \vdash i \uparrow \text{int} \Rightarrow \top} \text{ (tc-int-up)} \\
\frac{}{G \vdash i \downarrow T \Rightarrow T \doteq \text{int}} \text{ (tc-int-dn)} \\
\frac{}{G \vdash x \uparrow G(x) \Rightarrow \top} \text{ (tc-var-up)} \\
\frac{}{G \vdash x \downarrow T \Rightarrow T \doteq G(x)} \text{ (tc-var-dn)} \\
\frac{\Sigma(\text{op}) = \tau_1 * \dots * \tau_n \rightarrow \tau \quad G \vdash e_1 \downarrow \tau_1 \Rightarrow \Phi_1 \quad \dots \quad G \vdash e_n \downarrow \tau_n \Rightarrow \Phi_n}{G \vdash \text{op}(e_1, \dots, e_n) \uparrow \tau \Rightarrow \Phi_1 \wedge \dots \wedge \Phi_n} \text{ (tc-op-up)} \\
\frac{\Sigma(\text{op}) = \tau_1 * \dots * \tau_n \rightarrow \tau \quad G \vdash e_1 \downarrow \tau_1 \Rightarrow \Phi_1 \quad \dots \quad G \vdash e_n \downarrow \tau_n \Rightarrow \Phi_n}{G \vdash \text{op}(e_1, \dots, e_n) \downarrow T \Rightarrow \Phi_1 \wedge \dots \wedge \Phi_n \wedge T \doteq \tau} \text{ (tc-op-dn)} \\
\frac{G \vdash e \downarrow \text{bool} \Rightarrow \Phi \quad G \vdash e_1 \uparrow T \Rightarrow \Phi_1 \quad G \vdash e_2 \downarrow T \Rightarrow \Phi_2}{G \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \uparrow T \Rightarrow \Phi \wedge \Phi_1 \wedge \Phi_2} \text{ (tc-if-up)} \\
\frac{G \vdash e \downarrow \text{bool} \Rightarrow \Phi \quad G \vdash e_1 \downarrow T \Rightarrow \Phi_1 \quad G \vdash e_2 \downarrow T \Rightarrow \Phi_2}{G \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \downarrow T \Rightarrow \Phi \wedge \Phi_1 \wedge \Phi_2} \text{ (tc-if-dn)} \\
\frac{G \vdash e_1 \uparrow T_1 \Rightarrow \Phi_1 \quad G \vdash e_2 \uparrow T_2 \Rightarrow \Phi_2}{G \vdash \{e_1, e_2\} \uparrow T_1 * T_2 \Rightarrow \Phi_1 \wedge \Phi_2} \text{ (tc-tup-up)} \\
\frac{G \vdash e_1 \downarrow T_1 \Rightarrow \Phi_1 \quad G \vdash e_2 \downarrow T_2 \Rightarrow \Phi_2}{G \vdash \{e_1, e_2\} \downarrow T_1 * T_2 \Rightarrow \Phi_1 \wedge \Phi_2} \text{ (tc-tup-dn)} \\
\frac{G \vdash e_1 \uparrow T_1 \Rightarrow \Phi_1 \quad G \vdash e_2 \uparrow T_2 \Rightarrow \Phi_2}{G \vdash \{e_1, e_2\} \downarrow X \Rightarrow \Phi_1 \wedge \Phi_2 \wedge X \doteq T_1 * T_2} \text{ (tc-tup-dn-atom)}
\end{array}$$

Figure 9.1: Type reconstruction rules for generating types constraints (I)

$$\begin{array}{c}
\frac{G \vdash e \uparrow T_1 * T_2 \Rightarrow \Phi}{G \vdash e.1 \uparrow T_1 \Rightarrow \Phi} \text{ (tc-fst-up)} \\
\frac{G \vdash e \uparrow X \Rightarrow \Phi}{G \vdash e.1 \uparrow X_1 \Rightarrow \Phi \wedge X \doteq X_1 * X_2} \text{ (tc-fst-up-atom)} \\
\frac{G \vdash e \downarrow T * X \Rightarrow \Phi}{G \vdash e.1 \downarrow T \Rightarrow \Phi} \text{ (tc-fst-dn)} \\
\frac{G \vdash e \uparrow T_1 * T_2 \Rightarrow \Phi}{G \vdash e.2 \uparrow T_2 \Rightarrow \Phi} \text{ (tc-snd-up)} \\
\frac{G \vdash e \uparrow X \Rightarrow \Phi}{G \vdash e.2 \uparrow X_2 \Rightarrow \Phi \wedge X \doteq X_1 * X_2} \text{ (tc-snd-up-atom)} \\
\frac{G \vdash e \downarrow X * T \Rightarrow \Phi}{G \vdash e.2 \downarrow T \Rightarrow \Phi} \text{ (tc-snd-dn)} \\
\frac{G, f : X_1 \rightarrow X_2, x : X_1 \vdash e \downarrow X_2 \Rightarrow \Phi}{G \vdash \mathbf{fun} f(x) \text{ is } e \uparrow X_1 \rightarrow X_2 \Rightarrow \Phi} \text{ (tc-fun-up)} \\
\frac{G, f : T_1 \rightarrow T_2, x : T_1 \vdash e \downarrow T_2 \Rightarrow \Phi}{G \vdash \mathbf{fun} f(x) \text{ is } e \downarrow T_1 \rightarrow T_2 \Rightarrow \Phi} \text{ (tc-fun-dn)} \\
\frac{G \vdash \mathbf{fun} f(x) \text{ is } e \uparrow T \Rightarrow \Phi}{G \vdash \mathbf{fun} f(x) \text{ is } e \downarrow X \Rightarrow \Phi \wedge X \doteq T} \text{ (tc-fun-dn-atom)} \\
\frac{G, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow \Phi}{G \vdash \mathbf{fun} f(x : \tau_1) : \tau_2 \text{ is } e \uparrow \tau_1 \rightarrow \tau_2 \Rightarrow \Phi} \text{ (tc-fun-anno-up)} \\
\frac{G, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow \Phi}{G \vdash \mathbf{fun} f(x : \tau_1) : \tau_2 \text{ is } e \downarrow T \Rightarrow \Phi \wedge T = \tau_1 \rightarrow \tau_2} \text{ (tc-fun-anno-dn)} \\
\frac{G \vdash e_1 \uparrow T_1 \rightarrow T_2 \Rightarrow \Phi_1 \quad G \vdash e_2 \downarrow T_1 \Rightarrow \Phi_2}{G \vdash \mathbf{app}(e_1, e_2) \uparrow T_2 \Rightarrow \Phi_1 \wedge \Phi_2} \text{ (tc-app-up)} \\
\frac{G \vdash e_1 \uparrow X_1 \Rightarrow \Phi_1 \quad G \vdash e_2 \uparrow T \Rightarrow \Phi_2}{G \vdash \mathbf{app}(e_1, e_2) \uparrow X_2 \Rightarrow \Phi_1 \wedge \Phi_2 \wedge X_1 = T \rightarrow X_2} \text{ (tc-app-up-atom)} \\
\frac{G \vdash \mathbf{app}(e_1, e_2) \uparrow T_2 \Rightarrow \Phi}{G \vdash \mathbf{app}(e_1, e_2) \downarrow T_1 \Rightarrow \Phi \wedge T_1 \doteq T_2} \text{ (tc-app-dn)} \\
\frac{G \vdash e_1 \uparrow T_1 \Rightarrow \Phi_1 \quad G, x : T_1 \vdash e_2 \uparrow T_2 \Rightarrow \Phi_2}{G \vdash \mathbf{let} x = e_1 \text{ in } e_2 \uparrow T_2 \Rightarrow \Phi_1 \wedge \Phi_2} \text{ (tc-let-up)} \\
\frac{G \vdash e_1 \uparrow T_1 \Rightarrow \Phi_1 \quad G, x : T_1 \vdash e_2 \downarrow T_2 \Rightarrow \Phi_2}{G \vdash \mathbf{let} x = e_1 \text{ in } e_2 \downarrow T_2 \Rightarrow \Phi_1 \wedge \Phi_2} \text{ (tc-let-dn)} \\
\frac{G \vdash e \downarrow \tau \Rightarrow \Phi}{G \vdash (e : \tau) \uparrow \tau \Rightarrow \Phi} \text{ (tc-anno-up)} \\
\frac{G \vdash e \downarrow \tau \Rightarrow \Phi}{G \vdash (e : \tau) \downarrow T \Rightarrow \Phi \wedge \tau \doteq T} \text{ (tc-anno-dn)}
\end{array}$$

Figure 9.2: Type reconstruction rules for generating types constraints (II)

The type (τ_1, \dots, τ_n) is for an expression that defines n functions mutually recursively and these functions are of types τ_1, \dots, τ_n , respectively.

$$\begin{array}{c}
\Gamma, f_1 : \tau_1 \rightarrow \tau'_1, \dots, f_n : \tau_n \rightarrow \tau'_n, x_1 : \tau_1 \vdash e_1 : \tau'_1 \\
\vdots \\
\Gamma, f_1 : \tau_1 \rightarrow \tau'_1, \dots, f_n : \tau_n \rightarrow \tau'_n, x_n : \tau_n \vdash e_n : \tau'_n \\
\hline
\Gamma \vdash \mathbf{fun} \ f_1(x_1 : \tau_1) : \tau'_1 \ \mathbf{is} \ e_1 \ \mathbf{and} \dots \mathbf{and} \ \mathbf{fun} \ f_n(x_n : \tau_n) : \tau'_n \ \mathbf{is} \ e_n : (\tau_1 \rightarrow \tau'_1, \dots, \tau_n \rightarrow \tau'_n) \quad \mathbf{(type-funs)} \\
\hline
\Gamma \vdash e : (\tau_1, \dots, \tau_n) \quad 1 \leq i \leq n \\
\Gamma \vdash e \# i : \tau_i \quad \mathbf{(type-choose)}
\end{array}$$

$$\begin{array}{c}
\Gamma, f_1 : \tau_1 \rightarrow \tau'_1, \dots, f_n : \tau_n \rightarrow \tau'_n, x_1 : \tau_1 \vdash e_1 \downarrow \tau'_1 \Rightarrow \Phi_1 \\
\vdots \\
\Gamma, f_1 : \tau_1 \rightarrow \tau'_1, \dots, f_n : \tau_n \rightarrow \tau'_n, x_n : \tau_n \vdash e_n \downarrow \tau'_n \Rightarrow \Phi_n \\
\hline
\Gamma \vdash \mathbf{fun} \ f_1(x_1 : \tau_1) : \tau'_1 \ \mathbf{is} \ e_1 \ \mathbf{and} \dots \mathbf{and} \ \mathbf{fun} \ f_n(x_n : \tau_n) : \tau'_n \ \mathbf{is} \ e_n \uparrow (\tau_1 \rightarrow \tau'_1, \dots, \tau_n \rightarrow \tau'_n) \Rightarrow \Phi_1 \wedge \dots \wedge \Phi_n \quad \mathbf{(tc-funs-anno-up)}
\end{array}$$

$$\begin{array}{c}
\Gamma, f_1 : X_1 \rightarrow X'_1, \dots, f_n : X_n \rightarrow X'_n, x_1 : X_1 \vdash e_1 \downarrow X'_1 \Rightarrow \Phi_1 \\
\vdots \\
\Gamma, f_1 : X_1 \rightarrow X'_1, \dots, f_n : X_n \rightarrow X'_n, x_n : X_n \vdash e_n \downarrow X'_n \Rightarrow \Phi_n \\
\hline
\Gamma \vdash \mathbf{fun} \ f_1(x_1) \ \mathbf{is} \ e_1 \ \mathbf{and} \dots \mathbf{and} \ \mathbf{fun} \ f_n(x_n) \ \mathbf{is} \ e_n \uparrow (X_1 \rightarrow X'_1, \dots, X_n \rightarrow X'_n) \Rightarrow \Phi_1 \wedge \dots \wedge \Phi_n \quad \mathbf{(tc-funs-up)}
\end{array}$$

$$\frac{G \vdash e \uparrow (T_1, \dots, T_n) \Rightarrow \Phi \quad 1 \leq i \leq n}{G \vdash e \# i \uparrow T_i \Rightarrow \Phi} \quad \mathbf{(tc-choose-up)}$$

$$\frac{G \vdash e \uparrow (T_1, \dots, T_n) \Rightarrow \Phi \quad 1 \leq i \leq n}{G \vdash e \# i \downarrow T \Rightarrow \Phi \wedge T = T_i} \quad \mathbf{(tc-choose-dn)}$$

Notice that neither rule **(tc-funs-dn)** nor rule **(tc-funs-anno-dn)** is presented. There is probably no need for such rules in practice. In case there is such a need, we can always use either rule **(tc-funs-up)** or rule **(tc-funs-anno-up)** to synthesize a type and then form a type constraint between the synthesized type and the type being checked against.