Master Thesis

An extension of soft typing for static detection of runtime type errors

—実行時型エラーの静的検出のための柔らかい型の拡張—

350601151 Akihisa Yamada

Graduate School of Information Science, Nagoya University

Department of Computer Science and Mathematical Informatics

January 2008

Abstract

In a *dynamically typed language* like LISP, types are checked during program execution. The most important advantage over statically typed languages is its rich expressiveness. On the other hand, one of disadvantages is the difficulty in debugging. If type errors were detected before execution, i.e. *statically*, debugging would be easier.

Soft typing statically detects potential type errors for dynamically typed languages. This typing is 'soft' in the sense that any ill-typed programs are not rejected, so that expressiveness is not spoiled. A soft type system inserts a runtime check to ill-typed programs, since they may not cause an error for some input. In other words, programmers need to check if the program is executable for intended inputs.

Complete typing was introduced to make this check easier, and it indeed detects programs which always cause an error. A complete type system requires to decide type *disjointness*, but its efficient decision algorithm is not yet obtained.

Our ambition is to simulate complete typing, by extending soft typing. We add a special type E called "error type" to soft type system. The error type expresses the type of programs which always cause an error. So if a program has the type E, programmers can immediately see there is a bug. Thanks to the *union types* which already exist in soft typing, we do not loose the advantage of softness; if a program has a union type with E, we may insert a runtime check. We also enrich soft typing by introducing *intersection* types and *complement* types. These types are essential for the error type to work. For example, these extensions allow the typing $f : (int \to int) \cap (int^{\mathsf{C}} \to \mathsf{E})$ for a function f defined exactly on integers.

Our approach stays syntactic and does not go into denotational semantics

such as the *ideal model*. So we skip the profound discussion of 'contractivity' or 'well-formedness', and the soundness is shown by proving the *subject reduction* property. In order to avoid complicating this proof, we show that *transitivity* rule of our subtyping can be dropped. Additionally, we show that with a suitable type assignment of pre-defined functions, our system is *total*, that is, every expression has a type.

Contents

Chapter 1

Introduction

The notion of *typing* is widely used to assure that the result of program execution is defined. There are two major classes of typing, *static* typing and *dynamic* typing.

In a statically typed language like C, Java or ML, types are checked at compile time. They are generally *sound*, in the sense that result of an accepted program is defined, if it terminates. But they cannot be *complete*, so there may be meaningful ones in rejected programs. In order to make such programs typechecked, a careful modification may be needed, for example attaching tags to input, or adding branches for every possible types, or completely new structure of programs may be needed. However strong static typings may become, they cannot accept all the meaningful programs, since this is an undecidable problem.

In a dynamically typed language like LISP, types are checked during program execution. So the above problem does not happen, and every meaningful programs can be executed. As a positive result, a dynamically typed language is more expressive than statically typed languages. Programs are simpler (at least in source code), and more reusable in many context, provided they do not raise a runtime error for the given input.

As a negative result, on the other hand, programmers must by themselves assure that programs do not raise runtime errors for intended inputs. Also execution is not always effective since a runtime check occurs for every function application.

Soft typing[4] is a compromise between static and dynamic typing. Like static

typing, types are checked at compile time. But unlike static typing, a soft type system does not reject an ill-typed program, instead it inserts an explicit runtime check. So indeed every meaningful program is accepted, and expressiveness of dynamic typing is retained. But for debugging purpose it is not enough; programmers must assure that the inserted runtime check will not fail for intended inputs.

Complete typing[11] is a concept of typing which detects inevitable type errors. Debugging will become easier since 'completely' untyped programs is obviously a bug. But complete typing needs a decision of *type disjointness*, whose efficient algorithm is unfortunately not known[10].

The principal idea of this study is to introduce a special type E called an *error type*, which expresses the type of programs who always cause type errors. A program typed without E can be accepted (without a runtime check), so in this sense this typing is 'sound'. A program typed by E can be immediately rejected, so in this sense this typing is 'complete'. Of course a static typing cannot be sound and complete in a strict meaning, so our system may return 'unknown' for a program typed by a *union type*, which is already introduced in soft typing, with E.

Our extension also includes *intersection types* and *complement types*. An intersection type can be used to denote *overloading*, for example:

$$+ : (\mathrm{int} \times \mathrm{int} \to \mathrm{int}) \cap (\mathrm{real} \times \mathrm{real} \to \mathrm{real}).$$

The complement type is useful to denote *partial functions*, for example:

$$\mathtt{fact} : (\mathrm{posint} \to \mathrm{posint}) \cap (\mathrm{posint}^{\mathsf{C}} \to \mathsf{E}).$$

where posint stands for the type for positive integers. As the above example illustrates, the error type become much useful with intersections and complements.

In this thesis, we first present a generalization of functional programming language in terms of λ-*calculus*, combined with *rewriting systems*. Here a type error is defined as a *normal form* whose structure is not acceptable. Then we present a type language with its *subtyping* rules and *typing* rules. Our approach will not go into denotational semantics such as the *ideal model*[6]. Instead we

stay syntactic and define a typing hold if and only if there is a finite proof. And imitating [13], soundness is shown by proving the *subject reduction* property of our system.

To avoid complicating the proof, we exclude the *transitivity* rule from our subtyping rules. Though, we show that this is not a defect; our subtyping is already transitive.

Finally we show that with an appropriate type assignment for pre-defined functions, our system is *total*, in other words, every expression has a type.

The rest of this thesis is as follows: Chapter 2 gives preliminaries, where we generalize programs and types by a notion of *terms with variable binders*. Chapter 3 defines *expressions* as an idealized programming language. The β-reduction, pre-defined functions, user-defined functions, and type errors are formalized here. Chapter 4 defines types, subtyping, and typing. Some concluding soundness theorems are listed in Section 4.4. Totality is described in Section 4.5. Chapter 6 gives a conclusion.

Chapter 2

Preliminaries

2.1 Basic notations

For two mappings $f : A \to B$ and $g : B \to C$, their composition $g \cdot f$ is defined by $g \cdot f(a) = g(f(a))$.

Definition 2.1.1 For a mapping $f : A \to B$, $a \in A$ and $b \in B$, the updated mapping $f[a \mapsto b] : A \to B$ is defined by:

$$f[a \mapsto b](c) := \begin{cases} b & \text{if } c = a \\ f(c) & \text{otherwise} \end{cases}$$

For a relation $R \subseteq A \times B$, $\langle a, b \rangle \in R$ is abbreviated by $a\ R\ b$. For two relations R and S, their **composition** $R \circ S$ is defined by $R \circ S := \{\langle a, c \rangle \mid \exists b.\ a\ R\ b\ S\ c\}$.

2.2 Abstract reduction systems

Definition 2.2.1 An **abstract reduction system** (ARS) is a pair $\langle A, \longrightarrow \rangle$, where $\longrightarrow$ is a binary relation on the set A.

Definition 2.2.2 For an ARS $\langle A, \longrightarrow \rangle$, we define the following relations:

$$\begin{aligned} \xrightarrow{0} &:= \{\langle a, a\rangle \mid a \in A\} \\ \xrightarrow{i+1} &:= \xrightarrow{i} \circ \longrightarrow \\ \xrightarrow{*} &:= \bigcup_{0 \leq i} \xrightarrow{i} \end{aligned}$$

Definition 2.2.3 Following notions are defined for elements in A:

1. $a \in A$ is **reducible** iff there exists $b \in A$ and $a \longrightarrow b$.
2. a is in **normal form** iff a is not reducible. $\mathsf{NF}(\longrightarrow)$ denotes the set of all normal forms.
3. b is a **normal form of** a iff $a \xrightarrow{*} b$ and b is in normal form.

2.3 Terms with variable binders

Definition 2.3.1 Let $\mathcal{B}$ be set of **variable binders**, $\mathcal{F}$ be set of **function symbols**, and $\mathcal{V}$ be set of **variable symbols**. Also we assume the mapping $\mathsf{arity} : \mathcal{F} \to \mathbb{N}$ is given. The set $\mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V})$ of **terms with variable binders** is defined inductively as follows:

1. $\mathcal{V} \subseteq \mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V})$
2. $f(s_1, \cdots, s_n) \in \mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V})$ if $f \in \mathcal{F},\ n = \mathsf{arity}(f),\ s_1, \cdots, s_n \in \mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V})$
3. $bx.\, s \in \mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V})$ if $b \in \mathcal{B},\ x \in \mathcal{V},\ s \in \mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V})$

For $c \in \mathcal{F}$ with $\mathsf{arity}(c) = 0$, $c()$ is abbreviated by c. For $op \in \mathcal{F}$ with $\mathsf{arity}(op) = 2$, we may use infix notation: $s\ op\ t := op(s, t)$. Also for $b \in \mathcal{B}$, $bx_1.\cdots bx_n.\, s$ may be abbreviated by $bx_1 \cdots x_n.\ s$. Parenthesis may be used to avoid ambiguity.

Definition 2.3.2 For a term s, the set $\mathsf{FV}(s)$ of **free variables** and $\mathsf{BV}(s)$ of **bound variables** are defined as follows:

1. $\mathsf{FV}(x) = \{x\}, \quad \mathsf{BV}(x) = \emptyset \qquad \text{if } x \in \mathcal{V}$
2. $\mathsf{FV}(f(s_1, \cdots, s_n)) = \bigcup_{i \leq n} \mathsf{FV}(s_i)$,
   $\mathsf{BV}(f(s_1, \cdots, s_n)) = \bigcup_{i \leq n} \mathsf{BV}(s_i)$
3. $\mathsf{FV}(bx.\, s) = \mathsf{FV}(s) \setminus \{x\}, \quad \mathsf{BV}(bx.\, s) = \mathsf{BV}(s) \cup \{x\}$

A term s is **closed** iff there is no free variable in s.

Definition 2.3.3 A mapping $\theta : \mathcal{V} \to \mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V})$ is called a **substitution**, if the **domain** of θ, defined by $\mathsf{Dom}(\theta) := \{x \mid x \neq \theta(x)\}$, is finite. If $\mathsf{Dom}(\theta) = \{x_1, \cdots, x_n\}$, we denote θ by $[x_1 \mapsto \theta(x_1), \cdots, x_n \mapsto \theta(x_n)]$. The **range** of θ is $\mathsf{Ran}(\theta) := \theta(\mathsf{Dom}(\theta))$ and **free-variable range** of θ is $\mathsf{FVRan}(\theta) := \mathsf{FV}(\mathsf{Ran}(\theta))$.

We extend θ to $\hat{\theta} : \mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V}) \rightharpoonup \mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V})$ as follows:

1. $\hat{\theta}(x) = \theta(x) \quad \text{if } x \in \mathcal{X}$
2. $\hat{\theta}(f(s_1, \cdots, s_n)) = f(\hat{\theta}(s_1), \cdots, \hat{\theta}(s_n)) \quad \text{if } f \in \Sigma$
3. $\hat{\theta}(bx.\, s) = bx.\, \hat{\theta}(s) \quad \text{if } x \notin \mathsf{Dom}(\theta) \cup \mathsf{FVRan}(\theta)$

We abbreviate $\hat{\theta}(s)$ by $s\theta$.

Note that the extended substitutions are not total on $\mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V})$. This problem is later avoided by renaming bound variables by *α-conversion*.

Definition 2.3.4 $C \in \mathcal{U}(\mathcal{B}, \mathcal{F} \cup \{\Box\}, \mathcal{V})$ is a **context** iff it has a special constant $\Box$ at exactly one position. We write $C[s]$ denoting the term obtained by substituting s to $\Box$ in C.

Definition 2.3.5 A pair $\langle l, r \rangle \in \mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V})^2$ is a **rewrite rule** (a rule in short) and written $l \to r$, if $\mathsf{FV}(l) \supseteq \mathsf{FV}(r)$. A **rule set** R is a set of rules. The **rewrite relation at root position** induced by R is defined by the following:

$$s \xrightarrow[R]{}^{\varepsilon} t \iff^{\text{def}} \exists\theta.\ s = l\theta,\ t = r\theta,\ l \to r \in R.$$

General **rewrite relation** is defined by the following:

$$s \xrightarrow[R]{} t \iff^{\text{def}} \exists C.\ s = C[s'],\ t = C[t'],\ s' \xrightarrow[R]{}^{\varepsilon} t'.$$

Definition 2.3.6 The **α-conversion** is the relation $\xrightarrow[\alpha]{*}$, induced by the following rule set α:

$$\alpha := \{bx.\, s \to by.\, s[x \mapsto y] \mid s \in \mathcal{U}(\mathcal{B}, \mathcal{F}, \mathcal{V}),\ b \in \mathcal{B},\ x \in \mathcal{V},\ y \in \mathcal{V} \setminus \mathsf{FV}(s)\}.$$

In the latter of this thesis we assume α-convertible terms are equal, and so substitutions become total.

Chapter 3

Expressions

3.1 λ-calculus with pairs

Definition 3.1.1 (the syntax) Let $\mathcal{X}$ be the set of **expression variables** (variables, in short) and Σ be the set of **expression functions** (functions, in short) with $\forall f \in \Sigma.\ \mathsf{arity}(f) = 0$. The set Λ of **expressions** is defined by:

$$\Lambda := \mathcal{U}(\{\lambda\}, \{\mathsf{app}, \mathsf{pair}\} \cup \Sigma, \mathcal{X})$$

where app and pair are special function symbols with $\mathsf{arity}(\mathsf{app}) = \mathsf{arity}(\mathsf{pair}) = 2$. For $X \subseteq \mathcal{X}$, the set of expressions whose free variables are in X is denoted by $\Lambda(X) := \{s \in \Lambda \mid \mathsf{FV}(s) \subseteq X\}$.

In the latter, f ranges over Σ, x, y, z range over $\mathcal{X}$ and s, t, u range over Λ. We denote $\mathsf{app}(s, t)$ by st with left associativity, and denote $\mathsf{pair}(s, t)$ by (s, t). We also abbreviate $(s_1, (\cdots, (s_{n-1}, s_n)\cdots))$ by $(s_1, \cdots, s_{n-1}, s_n)$.

Now we define several rewrite relations. First one is the *β-reduction*.

Definition 3.1.2 The **β-reduction** is the rewrite relation $\underset{\beta}{\longrightarrow} \subseteq \Lambda^2$ induced by the following rule set β:

$$\beta := \{(\lambda x.\, s)t \to s[x \mapsto t] \mid s, t \in \Lambda\}.$$

In order to formalize type errors simply, we restrict pre-defined functions in the following way.

Definition 3.1.3 Let Σ_B be a set of **basic constants**. The set Λ_T of **tuple expressions** is defined as follows:

1. $\Sigma_B \cup \mathcal{X} \subseteq \Lambda_T$
2. $(p, q) \in \Lambda_T \quad \text{if } p, q \in \Lambda_T$

Assumption 3.1.4 (pre-defined functions) Let Σ_P be a set of **pre-defined** function symbols. We assume a rule set δ is given to define how pre-defined functions act, and for every rule $l \to r \in \delta$, l is in form fp with $f \in \Sigma_P$ and $p \in \Lambda_T$.

Note that the above restriction does not go along with *data constructors*. However, they can be encoded using pairs, for example $\texttt{succ}\ n \longrightarrow (\texttt{succ}', n)$.

Also note that pre-defined functions cannot be in *curried form*, but they also can be encoded using λ, for example $\texttt{add} \longrightarrow \lambda xy.\, \texttt{add}'(x, y)$.

Now we define a type error as a function application whose output is 'undefined'. Thanks to the above restriction, this formalization is a simple one.

Definition 3.1.5 Let $\mathsf{error} \in \Sigma$ be a new constant denoting a type error. The following rule set ϵ defines when a type error occurs:

$$\epsilon := \{(x, y)z \to \mathsf{error}\} \cup \{fv \to \mathsf{error} \mid v \in \Lambda(\emptyset), f \in \Sigma \text{ and } fv \in \mathsf{NF}(\beta \cup \delta)\}$$

Example 3.1.6 (list operations) Let $\Sigma_B = \{\texttt{[]}\} \cup \mathbb{N}$, $\Sigma_P = \{\texttt{hd}, \texttt{tl}, \texttt{::}\}$ and

$$\delta = \begin{cases} \texttt{hd}(x, y) \to x \\ \texttt{tl}(x, y) \to y \\ \texttt{::}(x, y) \to (x, y) \end{cases}$$

then for example the following rewriting sequence exists; here we use infix notation for :: with right associativity:

$$\begin{aligned} & \texttt{tl}(\texttt{hd}(\texttt{tl}(1\texttt{::}2\texttt{::[]}))) \\ \xrightarrow[\delta]{} \ & \texttt{tl}(\texttt{hd}(\texttt{tl}(1, 2\texttt{::[]}))) \\ \xrightarrow[\delta]{} \ & \texttt{tl}(\texttt{hd}(2\texttt{::[]})) \\ \xrightarrow[\delta]{} \ & \texttt{tl}(\texttt{hd}(2, \texttt{[]})) \\ \xrightarrow[\delta]{} \ & \texttt{tl}\ 2 \in \mathsf{NF}(\beta \cup \delta) \\ \xrightarrow[\epsilon]{} \ & \mathsf{error} \end{aligned}$$

Example 3.1.7 Let us consider the natural number constructor $\mathtt{succ}$, such that $\mathtt{succ}\ s$ causes an error if s is not in form $\mathtt{succ}^n\ 0$. The following encoding simulates this behavior well:

$$\delta_{\mathtt{succ}} = \begin{cases} \mathtt{succ}\ 0 & \to (\mathtt{succ}', 0) \\ \mathtt{succ}(\mathtt{succ}', x) & \to (\mathtt{succ}', (\mathtt{succ}', x)) \end{cases}$$

If s contains no $\mathtt{succ}'$, we can see that $\mathtt{succ}\ s \xrightarrow[\delta_{\mathtt{succ}} \cup \epsilon]{*} \mathsf{error}$ if and only if s is not in form $\mathtt{succ}^n\ 0$.

Definition 3.1.8 A (syntactic) **value** $v \in \Lambda_V$ is defined inductively as follows:

1. $\Sigma \subseteq \Lambda_V$
2. $(u, v) \in \Lambda_V \quad \text{if } u, v \in \Lambda_V$
3. $\lambda x.\, s \in \Lambda_V \quad \text{if } s \in \Lambda(\{x\})$

Lemma 3.1.9 Let s be a closed expression such that $s \in \mathsf{NF}(\xrightarrow[\beta \cup \delta \cup \epsilon]{})$. Then s is a value.

Proof Suppose $s \in \mathsf{NF}$ and $s = uv$. Since $u \in \mathsf{NF}$ we have $u \in \Lambda_V$ by the induction hypothesis.

Case $u = f \in \Sigma$. It contradicts because $fv \xrightarrow[\epsilon]{} \mathsf{error}$.

Case $u = (u_1, u_2)$. Same as above.

Case $u = \lambda x.\, u'$. It contradicts because $(\lambda x.\, u')v \xrightarrow[\beta]{} u'[x \mapsto v]$.

Thus s cannot be in form uv, and by induction s must be a value. □

The inverse will also hold if we restrict contexts to *evaluation contexts*, but in our study this is not needed.

Finally, we add *user-defined* functions:

Assumption 3.1.10 Let Σ_U be a set of **user-defined** symbols, which does not contain error. We assume the rule set $\gamma \subseteq \Sigma_U \times \Lambda(\emptyset)$ gives the definitions for Σ_U.

In the latter we consider $\Sigma = \Sigma_B \cup \Sigma_P \cup \Sigma_U \cup \{\mathsf{error}\}$, and abbreviate $\xrightarrow[\beta \cup \gamma \cup \delta \cup \epsilon]{}$ by $\longrightarrow$. It is obvious that Lemma 3.1.9 also holds for $\longrightarrow$, since every expression with a user-defined symbol is reducible with γ.

3.2 S-expressions

In this chapter we demonstrate that the language Λ is strong enough to simulate functional subset of a dynamically typed language. We choose Scheme as such a language and show how to encode a Scheme expression into a Λ expresion.

3.2.1 Syntax of S-expressions

Definition 3.2.1 Let $\texttt{()} \in \Sigma_P$. For a finite set $X \subseteq \mathcal{X}$ of variables, a **pattern** $p \in \mathrm{Pat}(X)$ is defined as follows:

1. $\texttt{()} \in \mathrm{Pat}(\emptyset)$
2. $x \in \mathrm{Pat}(\{x\})$
3. $\texttt{(}x\ \texttt{.}\ p\texttt{)} \in \mathrm{Pat}(X \cup \{x\}) \quad \text{if } x \notin X,\ p \in \mathrm{Pat}(X)$

We abbreviate $\texttt{(}x_1\ \texttt{.}\ \cdots \texttt{(}x_n\ \texttt{.}\ y\texttt{)} \cdots \texttt{)}$ by $\texttt{(}x_1\ \cdots\ x_n\ \texttt{.}\ y\texttt{)}$, and also by $\texttt{(}x_1\ \cdots\ x_n\texttt{)}$ if $y = \texttt{()}$. With $p \in \mathrm{Pat}$ and $f \in \Sigma$, the S-expression $e \in \mathcal{S}$ is defined as follows:

$$e ::= x \mid f \mid \texttt{(}e\ \cdots\ e\texttt{)} \mid \texttt{(lambda}\ p\ e\texttt{)}$$

Note that $\texttt{(}e\texttt{)} \neq e$.

3.2.2 Coding to expressions

Definition 3.2.2 (functions needed for coding)

$$\delta_c = \begin{cases} \texttt{hd}(x, y) \to x \\ \texttt{tl}(x, y) \to y \\ \texttt{checknil}(x, \texttt{()}) \to x \end{cases}$$

And we use following abbreviations.

$$\begin{aligned} \mathit{arg} &:= \lambda yz.\, y(\texttt{hd}\ z)(\texttt{tl}\ z) \\ \mathit{noarg} &:= \lambda xy.\, \texttt{checknil}(x, y) \end{aligned}$$

Lemma 3.2.3 Following reductions hold for arg and $noarg$:

1. $(arg\ \ \lambda x.\, s)(t,u) \xrightarrow[\beta\cup\delta_c]{*} s[x \mapsto t]u$
2. $noarg\ s\ \texttt{()} \xrightarrow[\beta\cup\delta_c]{*} s$
3. $(arg\ \ \lambda x_1.(\cdots(arg\ \ \lambda x_n.\, noarg\ s)\cdots))\ (s_1,\cdots,s_n,\texttt{()}) \xrightarrow[\beta\cup\delta_c]{*} s[x_1 \mapsto s_1,\cdots,x_n \mapsto s_n]$

Proof

1. As the following sequence shows

$$\begin{array}{rcl}(arg\ \ \lambda x.\, s)(t,u) & \xrightarrow[\beta]{} & (\lambda z.(\lambda x.\, s)(\texttt{hd}\ z)(\texttt{tl}\ z))\ (t,u) \\ & \xrightarrow[\beta]{} & (\lambda x.\, s)(\texttt{hd}(t,u))(\texttt{tl}(t,u)) \\ & \xrightarrow[\delta_c]{*} & (\lambda x.\, s)tu \\ & \xrightarrow[\beta]{} & s[x \mapsto t]u\end{array}$$

2. $noarg\ s\ \texttt{()} \xrightarrow[\beta]{} \texttt{checknil}(s,\texttt{()}) \xrightarrow[\delta_c]{} s$
3. Easy induction on n.

□

Definition 3.2.4 (coding) For an S-expression $e \in \mathcal{S}(\Sigma,\mathcal{X})$, the encoded expression $e^\Lambda \in \Lambda(\Sigma,\mathcal{X})$ is defined as follows:

- $a^\Lambda = a \quad \text{if } a \in \Sigma \cup \mathcal{X}$
- $\texttt{(}e_0\ e_1\ \cdots\ e_n\texttt{)}^\Lambda = e_0^\Lambda(e_1^\Lambda,\cdots,e_n^\Lambda,\texttt{()})$
- $\texttt{(lambda ()}\ e\texttt{)}^\Lambda = noarg\ e^\Lambda$
- $\texttt{(lambda}\ x\ e\texttt{)}^\Lambda = \lambda x.\, e^\Lambda$
- $\texttt{(lambda (}x\ \texttt{.}\ p\texttt{)}\ e\texttt{)}^\Lambda = arg\ \ \lambda x.\, \texttt{(lambda}\ p\ e\texttt{)}^\Lambda$

We abbreviate $arg\ \ \lambda x.\, \texttt{(lambda}\ p\ e\texttt{)}^\Lambda$ by $\lambda\texttt{(}x\ \texttt{.}\ p\texttt{)}.\, e^\Lambda$.

Example 3.2.5 (some functions of Scheme)

$$\delta_s = \begin{cases} \texttt{apply}(x, y, \texttt{()}) \to xy \\ \texttt{car}((x, y), \texttt{()}) \to x \\ \texttt{cdr}((x, y), \texttt{()}) \to y \\ \texttt{cons}(x, y, \texttt{()}) \to (x, y) \\ \texttt{list}\ x \to x \end{cases}$$

Example 3.2.6 (simulating the β-conversion of S-expressions) Following sequence simulates the β-conversion of S-expressions:

1. $\texttt{((lambda (}x_1\ \cdots\ x_n\texttt{)}\ e\texttt{)}\ e_1\ \cdots\ e_n\texttt{)}^{\Lambda} \xrightarrow{*} e^{\Lambda}[x_1 \mapsto e_1^{\Lambda}, \cdots, x_n \mapsto e_n^{\Lambda}]$

2. $\texttt{((lambda (}x_1\ \cdots\ x_n\ \texttt{.}\ y\texttt{)}\ e\texttt{)}\ e_1\ \cdots\ e_n\ d_1 \cdots d_m\texttt{)}^{\Lambda}$
$\xrightarrow{*} e^{\Lambda}[x_1 \mapsto e_1^{\Lambda}, \cdots, x_n \mapsto e_n^{\Lambda}, y \mapsto (d_1^{\Lambda}, \cdots, d_m^{\Lambda})]$

Chapter 4

Types

4.1 Syntax of the types

Definition 4.1.1 (syntax of the types) Let $\mathcal{V}$ be a set of **type variables** and $\mathcal{B}$ be a set of **base types** with $\forall \iota \in \mathcal{B}.\ \mathsf{arity}(\iota) = 0$. The set $\mathcal{T}$ of **type expressions** (types, in short) is defined by:

$$\mathcal{T} := \mathcal{U}(\{\mu, \pi\}, \{\mathsf{E}, \mathsf{C}, \times, \cap, \cup, \rightarrow\} \cup \mathcal{B}, \mathcal{V})$$

where $\mathsf{arity}(\mathsf{E}) = 0$, $\mathsf{arity}(\mathsf{C}) = 1$ and $\mathsf{arity}(op) = 2$ for $op = \times, \cap, \cup, \rightarrow$.

We abbreviate $\mathsf{C}(\sigma)$ by σ^{C} and use infix notation for $\times, \cap, \cup, \rightarrow$, and their priority is the order they were enumerated here. Associativities are right for $\times$ and $\rightarrow$, and left for $\cap$ and $\cup$.

In the latter ι ranges over base types, $\alpha, \beta, \gamma, \delta$ range over type variables, $\sigma, \tau, \rho, \upsilon$ range over types. Remember that renaming of bound variables are assumed to be equal.

4.2 Subtyping

Definition 4.2.1 A **subtyping** is a pair of types written $\sigma \subseteq \tau$.

A subtyping is a **variable subtyping** if its left side is a type variable, and a **variable supertyping** if its right side is a type variable. A variable subtyping $\alpha \subseteq \sigma$ and a supertyping $\sigma' \subseteq \alpha$ are **compatible** iff one of the followings holds:

1. $\sigma = \sigma'$,
2. $\sigma \in \mathcal{V}$ and $\sigma' \notin \mathcal{V}$, or
3. $\sigma \notin \mathcal{V}$ and $\sigma' \in \mathcal{V}$.

A **type constraint** Γ is a set of pairwise compatible variable subtypings and supertypings. Free type variables of Γ is defined by the following:

$$\mathsf{FV}(\Gamma) := \bigcup_{\sigma \subseteq \tau \in \Gamma} \mathsf{FV}(\sigma) \cup \mathsf{FV}(\tau)$$

For a variable subtyping or supertyping $\sigma \subseteq \tau$, we abbreviate $\Gamma \cup \{\sigma \subseteq \tau\}$ by $\Gamma, \sigma \subseteq \tau$ and $\alpha \subseteq \sigma, \sigma \subseteq \alpha$ by $\alpha \mapsto \sigma$. In the latter $\mathsf{FV}(a) \cup \mathsf{FV}(b)$ is abbreviated by $\mathsf{FV}(a, b)$.

Definition 4.2.2 (values of base types) For each base type ι, we assume the set $\Sigma_\iota \subseteq \Sigma_B$ of basic constants is given. Σ_ιc denotes the set $\Sigma_\iota^{\mathsf{C}} := \Lambda_V \setminus \Sigma_\iota$.

Definition 4.2.3 (subtyping rules) A subtyping $\sigma \subseteq \tau$ (syntactically) **holds** under a type constraint Γ iff $\Gamma \rhd \sigma \subseteq \tau$ has a finite proof using the following inference rules:

1. $\Gamma \rhd \sigma \subseteq \sigma$ (REF)
2. $\Gamma \rhd a \subseteq b$ (BASE) if a and b in form ι or ι^{C} and $\Sigma_a \subseteq \Sigma_b$
3. $\Gamma \rhd \iota \subseteq \mathsf{E}$ (ERR)
4. $\dfrac{\Gamma, \alpha \subseteq \sigma \rhd \sigma \subseteq \tau}{\Gamma, \alpha \subseteq \sigma \rhd \alpha \subseteq \tau}$ (LVAR)
5. $\dfrac{\Gamma, \tau \subseteq \beta \rhd \sigma \subseteq \tau}{\Gamma, \tau \subseteq \beta \rhd \sigma \subseteq \beta}$ (RVAR)
6. $\dfrac{\Gamma \rhd \rho \subseteq \tau \quad \Gamma \rhd \rho^{\mathsf{C}} \subseteq \tau}{\Gamma \rhd \sigma \subseteq \tau}$ (COMP)
7. $\dfrac{\Gamma \rhd \sigma_1 \subseteq \tau}{\Gamma \rhd \sigma_1 \cap \sigma_2 \subseteq \tau}$ (LI1) $\quad \dfrac{\Gamma \rhd \sigma_2 \subseteq \tau}{\Gamma \rhd \sigma_1 \cap \sigma_2 \subseteq \tau}$ (LI2)
8. $\dfrac{\Gamma \rhd \sigma \subseteq \tau_1 \quad \Gamma \rhd \sigma \subseteq \tau_2}{\Gamma \rhd \sigma \subseteq \tau_1 \cap \tau_2}$ (RI)

9. $\dfrac{\Gamma \rhd \sigma_1 \subseteq \tau \quad \Gamma \rhd \sigma_2 \subseteq \tau}{\Gamma \rhd \sigma_1 \cup \sigma_2 \subseteq \tau}$ (LU)

10. $\dfrac{\Gamma \rhd \sigma \subseteq \tau_1}{\Gamma \rhd \sigma \subseteq \tau_1 \cup \tau_2}$ (RU1) $\quad \dfrac{\Gamma \rhd \sigma \subseteq \tau_2}{\Gamma \rhd \sigma \subseteq \tau_1 \cup \tau_2}$ (RU2)

11. $\dfrac{\Gamma \rhd \sigma \subseteq \tau_1^{\mathsf{C}} \cap \tau_2^{\mathsf{C}}}{\Gamma \rhd \sigma \subseteq (\tau_1 \cup \tau_2)^{\mathsf{C}}}$ (CU)

12. $\dfrac{\Gamma \rhd \sigma \subseteq \tau_1^{\mathsf{C}} \cup \tau_2^{\mathsf{C}}}{\Gamma \rhd \sigma \subseteq (\tau_1 \cap \tau_2)^{\mathsf{C}}}$ (CI)

13. $\dfrac{\Gamma \rhd \sigma[\alpha \mapsto \rho] \subseteq \tau}{\Gamma \rhd \pi\alpha.\, \sigma \subseteq \tau}$ (INST) $\quad$ if $\alpha \notin \mathsf{FV}(\rho)$

14. $\dfrac{\Gamma, \alpha \mapsto \sigma \rhd \sigma \subseteq \tau}{\Gamma \rhd \mu\alpha.\, \sigma \subseteq \tau}$ (L-REC1) $\quad \dfrac{\Gamma, \beta \mapsto \tau \rhd \sigma \subseteq \tau}{\Gamma \rhd \sigma \subseteq \mu\beta.\, \tau}$ (R-REC1)

15. $\dfrac{\Gamma, \alpha \mapsto \sigma, \alpha \subseteq \beta, \beta \mapsto \tau \rhd \sigma \subseteq \tau}{\Gamma \rhd \mu\alpha.\, \sigma \subseteq \mu\beta.\, \tau}$ (REC2) $\quad$ if $\begin{cases} \alpha \in \mathsf{FV}(\sigma) \setminus \mathsf{FV}(\tau, \Gamma) \\ \beta \in \mathsf{FV}(\tau) \setminus \mathsf{FV}(\sigma, \Gamma) \end{cases}$

16. $\dfrac{\Gamma, \beta \mapsto \tau \rhd \sigma \subseteq \tau^{\mathsf{C}}}{\Gamma \rhd \sigma \subseteq (\mu\beta.\, \tau)^{\mathsf{C}}}$ (CREC) $\quad$ if $\beta \notin \mathsf{FV}(\sigma, \Gamma)$

17. $\dfrac{\Gamma \rhd \tau_1 \subseteq \sigma_1 \quad \Gamma \rhd \sigma_2 \subseteq \tau_2}{\Gamma \rhd \sigma_1 \to \sigma_2 \subseteq \tau_1 \to \tau_2}$ (FUN)

18. $\dfrac{\Gamma \rhd \tau_1 \subseteq \sigma_1 \quad \sigma_2 \subseteq \tau_2^{\mathsf{C}}}{\Gamma \rhd \sigma_1 \to \sigma_2 \subseteq (\tau_1 \to \tau_2)^{\mathsf{C}}}$ (C-FUN)

19. $\dfrac{\Gamma \rhd \sigma_1 \subseteq \tau_1 \quad \Gamma \rhd \sigma_2 \subseteq \tau_2}{\Gamma \rhd \sigma_1 \times \sigma_2 \subseteq \tau_1 \times \tau_2}$ (PROD)

20. $\dfrac{\Gamma \rhd \sigma_1 \subseteq \tau_1^{\mathsf{C}}}{\Gamma \rhd \sigma_1 \times \sigma_2 \subseteq (\tau_1 \times \tau_2)^{\mathsf{C}}}$ (C-P1) $\quad \dfrac{\Gamma \rhd \sigma_2 \subseteq \tau_2^{\mathsf{C}}}{\Gamma \rhd \sigma_1 \times \sigma_2 \subseteq (\tau_1 \times \tau_2)^{\mathsf{C}}}$ (C-P2)

21. $\dfrac{\Gamma \rhd \mathsf{E} \subseteq \rho}{\Gamma \rhd \iota \subseteq \tau \to \rho}$ (BF) $\quad \dfrac{\Gamma \rhd \mathsf{E} \subseteq \rho}{\Gamma \rhd \sigma_1 \times \sigma_2 \subseteq \tau \to \rho}$ (PF)

22. $\dfrac{\Gamma \rhd \sigma_2 \subseteq \mathsf{E}^{\mathsf{C}}}{\Gamma \rhd \sigma_1 \to \sigma_2 \subseteq \iota^{\mathsf{C}}}$ (C-FB) $\quad \dfrac{\Gamma \rhd \sigma_2 \subseteq \mathsf{E}^{\mathsf{C}}}{\Gamma \rhd \sigma_1 \to \sigma_2 \subseteq (\tau_1 \times \tau_2)^{\mathsf{C}}}$ (C-FP)

23. $\dfrac{\Gamma \rhd \sigma \subseteq \pi\alpha\beta.\, \alpha \times \beta}{\Gamma \rhd \sigma \subseteq \iota^{\mathsf{C}}}$ (C-PB) $\quad \dfrac{\Gamma \rhd \sigma \subseteq \iota}{\Gamma \rhd \sigma \subseteq (\tau_1 \times \tau_2)^{\mathsf{C}}}$ (C-BP)

Lemma 4.2.4 If $\Gamma \rhd \sigma \subseteq \tau$ has a proof, then $\Gamma \cup \Gamma' \rhd \sigma \subseteq \tau$ has a proof with the same structure.

Proof Obvious. □

Lemma 4.2.5 Let $\alpha \in \mathsf{FV}(\sigma) \setminus \mathsf{FV}(\tau, \rho, \Gamma)$. If $\Gamma, \sigma \subseteq \alpha, \alpha \subseteq \beta \rhd \tau \subseteq \rho$ then $\Gamma \rhd \tau \subseteq \rho$.

Proof Though (L-VAR) possibly deletes α, it is only if $\alpha \notin \mathsf{FV}(\sigma)$. The other rules where a variable may disappear are (L-REC1), (R-REC1), (CREC) and (REC2). None of them, though, leave that variable in the type constraint. Thus $\sigma \subseteq \alpha$ cannot be used in the proof. Only (R-VAR) uses $\alpha \subseteq \beta$ and it leaves α free. Thus $\alpha \subseteq \beta$ cannot be used, either. □

Note that we excluded the *transition* rule from our system:

$$\frac{\Gamma \rhd \sigma \subseteq \tau \qquad \Gamma \rhd \tau \subseteq \rho}{\Gamma \rhd \sigma \subseteq \rho} \text{ (TRANS)}$$

in order to make the latter discussion simpler. We can see, however, our system does not need it; it is already *transitive*:

Lemma 4.2.6 (transitivity) If $\Gamma \rhd \sigma \subseteq \tau$ (a) and $\Gamma \rhd \tau \subseteq \rho$ (b) then $\Gamma \rhd \sigma \subseteq \rho$.

Proof First, we show if the final step of (b) was (REF), (R-VAR), (RU1), (RU2), (RI), (R-REC1), (CU), (CI), (CREC), (C-PB) or (C-BP). In all these cases we have $\Gamma \rhd \tau \subseteq \rho'$ one step above for some appropriate ρ'. By the induction hypothesis we get $\Gamma \rhd \sigma \subseteq \rho'$, and applying the last rule we get our goal. Second, if (b) was obtained by (COMP), we have υ with $\Gamma \rhd \upsilon \subseteq \rho$ and $\Gamma \rhd \upsilon^{\mathsf{C}} \subseteq \rho$. So applying (COMP) with appropriate σ we get our goal.

So we prove other cases by induction on first (a) and then (b).

Case (REF), (LVAR), (LU), (LI1), (LI2), (INST), (L-REC1) or (COMP). Obvious from the induction hypothesis.

Case (BASE) is trivial.

Case (ERR). We have $\tau = \mathsf{E}$, but then (b) cannot hold.

Case (R-VAR). We have $\tau = \beta \in \mathcal{V},\ \tau' \subseteq \beta \in \Gamma$ and $\Gamma \rhd \sigma \subseteq \tau'$ (a'). The only candidate for (b) is (L-VAR), so there exist $\beta \subseteq \rho \in \Gamma$, and by compatibility $\rho = \tau'$ or $\rho = \gamma \in \mathcal{V}$. If $\rho = \tau'$, (a') is identical to our goal. If $\rho = \gamma$ we have the following proof:

$$\dfrac{\dfrac{\Gamma \rhd \tau' \subseteq \tau' \ \text{(REF)}}{\Gamma \rhd \tau' \subseteq \beta}\ \text{(R-VAR)}}{\Gamma \rhd \tau' \subseteq \gamma}\ \text{(R-VAR)}$$

Here we can apply the induction hypothesis with (a') and get $\Gamma \rhd \sigma \subseteq \gamma$.

Case (RU1) or (RU2). We have $\tau = \tau_1 \cup \tau_2$ and $\Gamma \rhd \sigma \subseteq \tau_i$ for some $i \in \{1, 2\}$. The structure of τ restricts the candidate of (b) to (LU). So we have $\Gamma \rhd \tau_i \subseteq \rho$ for the above i, and by the induction hypothesis we get $\Gamma \rhd \sigma \subseteq \rho$.

Case (RI). We have $\tau = \tau_1 \cap \tau_2$ and $\Gamma \rhd \sigma \subseteq \tau_i$ for either $i \in \{1, 2\}$. The structure of τ restricts the candidates of (b) to (LI1) and (LI2). So we have $\Gamma \rhd \tau_j \subseteq \rho$ for some $j \in \{1, 2\}$, and by the induction hypothesis of τ_j we get $\Gamma \rhd \sigma \subseteq \rho$.

Case (R-REC1). We have $\tau = \mu\beta.\, \tau'$ and $\Gamma, \beta \mapsto \tau' \rhd \sigma \subseteq \tau'$. There are only two candidates for (b).

- **Case** (L-REC1). We have $\Gamma, \beta \mapsto \tau' \rhd \tau' \subseteq \rho$. By the induction hypothesis we have $\Gamma, \beta \mapsto \tau' \rhd \sigma \subseteq \rho$. By Lemma 4.2.5 we get $\Gamma \rhd \sigma \subseteq \rho$.
- **Case** (REC2). We have $\rho = \mu\gamma.\, \rho'$ and $\Gamma, \beta \mapsto \tau', \beta \subseteq \gamma, \gamma \mapsto \rho' \rhd \tau' \subseteq \rho'$. Here let $\Gamma' = (\Gamma, \beta \mapsto \tau', \beta \subseteq \gamma, \gamma \mapsto \rho')$. By Lemma 4.2.4 we get $\Gamma' \rhd \sigma \subseteq \tau'$ and $\Gamma' \rhd \tau' \subseteq \rho'$, without increasing the heights. Thus the induction hypothesis can be applied and get $\Gamma' \rhd \sigma \subseteq \rho'$. Then by Lemma 4.2.5 we get $\Gamma, \gamma \mapsto \rho' \rhd \sigma \subseteq \rho'$. Now apply (R-REC1) and get $\Gamma \rhd \sigma \subseteq \mu\gamma.\, \rho'$.

Case (PROD). We have $\sigma = \sigma_1 \times \sigma_2,\ \tau = \tau_1 \times \tau_2$ and $\Gamma \rhd \sigma_i \subseteq \tau_i$ for either $i \in \{1, 2\}$. The candidates are (PROD) and (C-P1) and (C-P2).

- **Case** (PROD). We have $\rho = \rho_1 \times \rho_2$ and $\Gamma \rhd \tau_i \subseteq \rho_i$ for either $i \in \{1, 2\}$. By the induction hypothesis we get $\Gamma \rhd \sigma_1 \subseteq \rho_1$ and $\Gamma \rhd \sigma_2 \subseteq \rho_2$. Now apply (PROD) and get $\Gamma \rhd \sigma_1 \times \sigma_2 \subseteq \rho_1 \times \rho_2$.

Case (C-P1) or (C-P2). We have $\rho = (\rho_1 \times \rho_2)^{\mathsf{C}}$ and $\Gamma \rhd \tau_i \subseteq \rho_i^{\mathsf{C}}$ for some $i \in \{1, 2\}$. By the induction hypothesis we get $\Gamma \rhd \sigma_i \subseteq \rho_i^{\mathsf{C}}$, and by applying (C-P$i$), $\Gamma \rhd \sigma_1 \times \sigma_2 \subseteq (\rho_1 \times \rho_2)^{\mathsf{C}}$.

Case (FUN). We have $\sigma = \sigma_1 \to \sigma_2$, $\tau = \tau_1 \times \tau_2$ with $\Gamma \rhd \tau_1 \subseteq \sigma_1$ and $\Gamma \rhd \sigma_2 \subseteq \tau_2$. The candidates for (b) are (FUN) and (C-FUN).

Case (FUN). We have $\rho = \rho_1 \to \rho_2$, $\Gamma \rhd \rho_1 \subseteq \tau_1$ and $\Gamma \rhd \tau_2 \subseteq \rho_2$. By the induction hypothesis we get $\Gamma \rhd \rho_1 \subseteq \sigma_1$ and $\Gamma \rhd \sigma_2 \subseteq \rho_2$. Now apply (FUN) and get $\Gamma \rhd \sigma_1 \to \sigma_2 \subseteq \rho_1 \to \rho_2$.

Case (C-FUN). We have $\rho = (\rho_1 \to \rho_2)^{\mathsf{C}}$, $\Gamma \rhd \rho_1 \subseteq \tau_1$ and $\Gamma \rhd \tau_2 \subseteq \rho_2^{\mathsf{C}}$. By the induction hypothesis we get $\Gamma \rhd \rho_1 \subseteq \sigma_1$ and $\Gamma \rhd \sigma_2 \subseteq \rho_2^{\mathsf{C}}$. Now apply (C-FUN) and get $\Gamma \rhd \sigma_1 \to \sigma_2 \subseteq (\rho_1 \to \rho_2)^{\mathsf{C}}$.

Case (REC2). We have $\sigma = \mu\alpha.\, \sigma'$, $\tau = \mu\beta.\, \tau'$ and $\Gamma' \rhd \sigma' \subseteq \tau'$, where $\Gamma' = \Gamma, \alpha \mapsto \sigma', \alpha \subseteq \beta, \beta \mapsto \tau'$. The only candidate for (b) is (REC2) and we get $\rho = \mu\gamma.\, \rho'$ and $\Gamma, \beta \mapsto \tau', \beta \subseteq \gamma, \gamma \mapsto \rho' \rhd \tau' \subseteq \rho'$. Here let $\Gamma'' = (\Gamma', \beta \subseteq \gamma, \gamma \mapsto \rho')$. By Lemma 4.2.4 we get $\Gamma'' \rhd \sigma' \subseteq \tau'$ and $\Gamma'' \rhd \tau' \subseteq \rho'$, without increasing the heights. Thus the induction hypothesis can be applied and get $\Gamma'' \rhd \sigma' \subseteq \rho'$. By Lemma 4.2.5 we get $\Gamma, \alpha \mapsto \sigma', \alpha \subseteq \gamma, \gamma \mapsto \rho' \rhd \sigma' \subseteq \rho'$. Now apply (REC2) and get $\Gamma \rhd \pi\alpha.\, \sigma' \subseteq \pi\gamma.\, \rho'$.

Case (CU), (CI), (C-FUN), (C-P1), (C-P2), (C-FP), (C-BP) or (CREC). There is no candidate left for (b).

Case (C-FB) or (C-PB). We have $\tau = \iota^{\mathsf{C}}$. The only candidate for (b) is (BASE). Since $\Sigma_\iota^{\mathsf{C}} \subseteq \Sigma_{\iota'}$ cannot hold for any ι', we have some ι' with $\rho = \iota'^{\mathsf{C}}$. Applying (C-PB) we get $\sigma \subseteq \iota'^{\mathsf{C}}$.

Case (BF) or (PF). We have $\tau = \tau_1 \to \tau_2$ and $\Gamma \rhd \mathsf{E} \subseteq \tau_2$. The candidates for (b) are (FUN) and (C-FUN).

Case (FUN). We have $\rho = \rho_1 \to \rho_2$, $\Gamma \rhd \rho_1 \subseteq \tau_2$ and $\Gamma \rhd \tau_2 \subseteq \rho_2$. By the induction hypothesis we get $\Gamma \rhd \mathsf{E} \subseteq \rho_2$. So applying (BF) or (PF) we get our goal.

Case (C-FUN). We have $\rho = (\rho_1 \to \rho_2)^{\mathsf{C}}$, $\Gamma \rhd \rho_1 \subseteq \tau_1$ and $\Gamma \rhd \tau_2 \subseteq \rho_2^{\mathsf{C}}$. By the induction hypothesis we get $\Gamma \rhd \mathsf{E} \subseteq \rho_2^{\mathsf{C}}$. So by applying (BF) or (PF) we get our goal.

□

4.3 Type judgments

Definition 4.3.1 A **type judgment** is a pair of an expression s and a type σ, written $s : \sigma$. A **type environment** Δ is a partial mapping $\Sigma_P \cup \Sigma_U \cup \mathcal{X} \rightharpoonup \mathcal{T}$.

For $x \in \Sigma \cup \mathcal{X}$ and $\sigma \in \mathcal{T}$, $\Delta[x \mapsto \sigma]$ is abbreviated by $\Delta, x : \sigma$. Free variable range is defined by $\mathsf{FVRan}(\Delta) := \mathsf{FV}(\mathsf{Ran}(\Delta))$.

Definition 4.3.2 (typing rules) A judgment $s : \sigma$ syntactically holds under a type constraint Γ and a type environment Δ iff $\Gamma, \Delta \rhd s : \sigma$ has a finte proof using the following rules:

1. $\Gamma, \Delta \rhd c : \iota$ (BASE) if $c \in \Sigma_\iota$
2. $\Gamma, \Delta \rhd \mathsf{error} : \mathsf{E}$ (ERR)
3. $\Gamma, \Delta \rhd x : \Delta(x)$ (ASSUMP) if $x \in \mathsf{Dom}(\Delta)$
4. $\dfrac{\Gamma, \Delta, x : \tau \rhd u : \rho}{\Gamma, \Delta \rhd \lambda x.\, u : \tau \to \rho}$ (ABST) if $x \notin \mathsf{Dom}(\Delta)$
5. $\dfrac{\Gamma, \Delta \rhd s : \tau \to \sigma \quad \Gamma, \Delta \rhd t : \tau}{\Gamma, \Delta \rhd st : \sigma}$ (APP)
6. $\dfrac{\Gamma, \Delta \rhd s : \sigma \quad \Gamma, \Delta \rhd t : \tau}{\Gamma, \Delta \rhd (s, t) : \sigma \times \tau}$ (PAIR)
7. $\dfrac{\Gamma, \Delta \rhd s : \sigma \quad \Gamma \rhd \sigma \subseteq \sigma'}{\Gamma, \Delta \rhd s : \sigma'}$ (SUB)
8. $\dfrac{\Gamma, \Delta \rhd s : \sigma}{\Gamma, \Delta \rhd s : \pi\alpha.\, \sigma}$ (GEN) if $\alpha \notin \mathsf{FV}(\Gamma) \cup \mathsf{FVRan}(\Delta)$
9. $\dfrac{\Gamma, \Delta \rhd s : \sigma_1 \quad \Gamma, \Delta \rhd s : \sigma_2}{\Gamma, \Delta \rhd s : \sigma_1 \cap \sigma_2}$ (INT)

The following lemma states that an instantiation of an expression preserves its type as far as the substitution does not conflict with the type environment:

Lemma 4.3.3 (substitution) Let $y \notin \mathsf{Dom}(\Delta)$. If $\Gamma, \Delta, y : \tau \triangleright s : \sigma$ (a) and $\Gamma, \Delta \triangleright t : \tau$ (b) then $\Gamma, \Delta \triangleright s[y \mapsto t] : \sigma$.

Proof Let $\Delta' = (\Delta, y : \tau)$. Proof proceeds by induction on (a).

Case (BASE) or (ERR). Immediate from the induction hypothesis since $s[y \mapsto t] = s$.

Case (ASSUMP). We have $s = x \in \mathcal{X} \cup \Sigma$ and $\sigma = \Delta'(x)$.
It is obvious if $x \neq y$. If $x = y$ then we get $\sigma = \tau$ and $s[y \mapsto t] = t$. So the second precondition is identical to our goal.

Case (ABST). We have $s = \lambda x.\, u$ and $\Gamma, \Delta', x : \sigma' \triangleright u : \rho$.
If $x = y$ then $s[y \mapsto t] = s$ and we can get a proof of $\Gamma, \Delta \triangleright s : \sigma$ in the same structure with that of (a).
Suppose $x \neq y$. Then by the induction hypothesis we have $\Gamma, \Delta, x : \sigma' \triangleright u[y \mapsto t] : \rho$. By (ABST) and the fact $(\lambda x.\, u)[y \mapsto t] = \lambda x.\, u[y \mapsto t]$, we get $\Gamma, \Delta \triangleright (\lambda x.\, u)[y \mapsto t] : \sigma' \to \rho$.

Case (APP). We have $s = s'u$, $\Gamma, \Delta' \triangleright s' : \rho \to \tau$ and $\Gamma, \Delta' \triangleright u : \rho$.
By the induction hypothesis we have $\Gamma, \Delta \triangleright s'[y \mapsto t] : \rho \to \tau$ and $\Gamma, \Delta \triangleright u[y \mapsto t] : \rho$. By (APP) and the fact that $(s'u)[y \mapsto t] = s'[y \mapsto t]\, u[y \mapsto t]$, we get $\Gamma, \Delta \triangleright (s'u)[y \mapsto t] : \tau$.

Case (PAIR). We have $s = (s_1, s_2)$, $\sigma = \sigma_1 \times \sigma_2$, $\Gamma, \Delta' \triangleright s_1 : \sigma_1$ and $\Gamma, \Delta' \triangleright s_2 : \sigma_2$.
By the induction hypothesis we have $\Gamma, \Delta \triangleright s_1[y \mapsto t] : \sigma_1$ and $\Gamma, \Delta \triangleright s_2[y \mapsto t] : \sigma_1$. By (PAIR) and the fact that $(s_1, s_2)[y \mapsto t] = (s_1[y \mapsto t], s_2[y \mapsto t])$ we get $\Gamma, \Delta \triangleright (s_1, s_2)[y \mapsto t] : \sigma_1 \times \sigma_2$.

Case (SUB). We have $\Gamma \triangleright \sigma' \subseteq \sigma$ and $\Gamma, \Delta' \triangleright s : \sigma'$.
By the induction hypothesis we have $\Gamma, \Delta \triangleright s[y \mapsto t] : \sigma'$. By (SUB) we get $\Gamma, \Delta \triangleright s[y \mapsto t] : \sigma$.

Case (GEN). We have $\sigma = \pi\alpha.\,\sigma'$, $\alpha \notin \mathsf{FV}(\Delta')$ and $\Gamma, \Delta' \triangleright s : \sigma'$.
By induction hypothesis we have $\Gamma, \Delta \triangleright s[y \mapsto t] : \sigma'$. By (GEN) we get $\Gamma, \Delta \triangleright s[y \mapsto t] : \pi\alpha.\,\sigma'$.

Case (INT). We have $\sigma = \sigma_1 \cap \sigma_2$, $\Gamma, \Delta' \triangleright s : \sigma_1$ and $\Gamma, \Delta' \triangleright s : \sigma_2$.
By the induction hypothesis we have $\Gamma, \Delta \triangleright s[y \mapsto t] : \sigma_1$ and $\Gamma, \Delta \triangleright s[y \mapsto t] : \sigma_1$. By (INT) we get $\Gamma, \Delta \triangleright s[y \mapsto t] : \sigma_1 \cap \sigma_2$.

□

The following lemma states that a typing preserves under a 'stronger' environment:

Lemma 4.3.4 If $\Gamma, \Delta, y : \tau \triangleright s : \sigma$ (a) and $\Gamma \triangleright \tau' \subseteq \tau$ (b) then $\Gamma, \Delta, y : \tau' \triangleright s : \sigma$.

Proof By induction on (a).

Case (ASSUMP). We have $s = x \in \mathcal{X}$ and if $x \neq y$, $\Delta(x) = \sigma$. So this case is obvious. If $x = y$, then $\sigma = \tau$ and we have the following proof:

$$\frac{\Gamma, \Delta, y : \tau' \triangleright y : \tau' \quad \text{(ASSUMP)} \quad \Gamma \triangleright \tau' \subseteq \tau}{\Gamma, \Delta, y : \tau' \triangleright y : \tau} \text{(SUB)}$$

This is identical to our goal.

Case Others are obvious from the induction hypothesis, since they do not depend on Δ.

□

With the above lemma we can show the following which states that a type of an abstraction is always 'functional', in the sense that it is not a base type or a product type:

Lemma 4.3.5 If $\Gamma \triangleright \rho \subseteq \iota$ (a) or $\Gamma \triangleright \rho \subseteq \sigma_1 \times \sigma_2$ (a') then $\Gamma, \Delta \triangleright \lambda x.\, s : \rho$ (b) cannot hold.

Proof By induction on (a) and (a').

Case (REF), (BASE) or (PROD). Since (ABST) cannot be applied for (b), obvious from the induction hypothesis.

Case (L-VSUP). We must have $\alpha \supseteq \rho \in \Gamma$ and $\Gamma \rhd \alpha \subseteq \iota$ or $\sigma_1 \times \sigma_2$ (c), but no rule can be applied for (c).

Case (LI1) or (LI2). We have $\rho = \rho_1 \cap \rho_2$ and $\Gamma \rhd \rho_i \subseteq \iota$ or $\sigma_1 \times \sigma_2$ for some $i \in \{1, 2\}$. By the structure of the type, there are only two rules left for (b):

Case (INT). By the induction hypothesis $\Gamma, \Delta \rhd \lambda x.\, s : \rho_i$ cannot hold, so (INT) cannot be applied.

Case (SUB). We have $\Gamma \rhd \rho' \subseteq \rho_1 \cap \rho_2$. By Lemma 4.2.6 we have $\Gamma \rhd \rho' \subseteq \iota$ or $\sigma_1 \times \sigma_2$. Thus by the induction hypothesis $\Gamma, \Delta \rhd \lambda x.\, s : \rho'$ cannot hold, so (SUB) also cannot be applied.

Case (COMP). We need $\Gamma \rhd \tau \subseteq \iota$ or $\sigma_1 \times \sigma_2$ and $\Gamma \rhd \tau^{\mathsf{C}} \subseteq \iota$ or $\sigma_1 \times \sigma_2$ (c), and there is no rule but (BASE) applicable for (c), thus $\sigma_1 \times \sigma_2$ is impossible and τ must be a base type. However, $\Sigma_\tau \subseteq \Sigma_\iota$ and $\Sigma_\tau^{\mathsf{C}} \subseteq \Sigma_\iota$ are inconsistent. Overall this case is not possible.

Case (INST). We have $\rho = \pi\alpha.\, \rho'$, $\alpha \notin \mathsf{FV}(\Gamma)$ and $\Gamma \rhd \rho' \subseteq \iota$ or $\sigma_1 \times \sigma_2$. By the structure of the type, there are only two rules left for (b):

Case (GEN). By the induction hypothesis we have $\Gamma \rhd \lambda x.\, s : \rho'$ cannot hold, so (GEN) cannot be applied.

Case (SUB). Since $\rho' = \rho'[\alpha \mapsto \alpha]$, we get the following proof:

$$\frac{\Gamma \rhd \rho'[\alpha \mapsto \alpha] \subseteq \iota \text{ or } \sigma_1 \times \sigma_2}{\Gamma \rhd \pi\alpha.\, \rho' \subseteq \iota \text{ or } \sigma_1 \times \sigma_2}\ (\text{INT})$$

Thus by the induction hypothesis $\Gamma \rhd \lambda x.\, s : \pi\alpha.\, \rho'$ cannot hold.

Case Others are not applicable.

□

With the above lemma, we can show that every function type of an abstraction can be inferred directly by (ABST). We will prove this in two steps:

Lemma 4.3.6 If $\Gamma, \Delta \rhd \lambda x.\, t : \rho$ (a) and $\Gamma \rhd \rho \subseteq \sigma \to \tau$ (b) then $\Gamma, \Delta, x : \sigma \rhd t : \tau$.

Proof By induction on (b).

Case (REF). We have $\rho = \sigma \to \tau$. This case is done by induction on (a).

Case (BASE), (ERR), (ASSUMP), (APP) or (PAIR). They are not applicable.

Case (ABST). We have $\Gamma, \Delta, x : \sigma \rhd t : \tau$ and this is what we want.

Case (SUB). We have ρ' with $\Gamma, \Delta \rhd \lambda x.\, t : \rho'$ and $\Gamma \rhd \rho' \subseteq \rho$. Since $\rho = \sigma \to \tau$, the induction hypothesis is identical to our goal.

Case (GEN) or (INT). Obvious from the induction hypothesis.

Case (BF) or (PF). We have $\rho = \iota$ or $\rho_1 \times \rho_2$, which are denied by Lemma 4.3.5.

Case (LVAR), (LU), (LI1), (LI2) or (INST). Obvious from the induction hypothesis.

Case (FUN). We have $\rho = \sigma' \to \tau'$, $\Gamma \rhd \sigma \subseteq \sigma'$ and $\Gamma \rhd \tau' \subseteq \tau$. By the induction hypothesis we have $\Gamma, \Delta, x : \sigma' \rhd t : \tau'$. By Lemma 4.3.4 we get $\Gamma, \Delta, x : \sigma \rhd t : \tau'$. Applying (SUB) we get our goal.

Case Others are not applicable.

□

Lemma 4.3.7 (abstraction) If $\Gamma, \Delta \rhd \lambda x.\, t : \sigma \to \tau$ then $\Gamma, \Delta, x : \sigma \rhd t : \tau$.

Proof Because $\Gamma \rhd \sigma \to \tau \subseteq \sigma \to \tau$ by (REF), it is immediate from Lemma 4.3.6. □

The substitution and the abstraction lemmas are enough for proving type preservation of the β-reduction. We need to give some assumptions to show that property of other reductions.

Assumption 4.3.8 (γ-typability) For $f \in \Sigma_U$ and $f \to s \in \gamma$, we assume that $\Delta(f) = \sigma$ implies $\Gamma, \Delta \rhd s : \sigma$.

Assumption 4.3.9 (δ-typability) For all $f \in \Sigma_P$ and an expression t, we assume $\Gamma, \Delta \rhd ft : \sigma$ and $ft \xrightarrow[\delta\cup\epsilon]{\varepsilon} s$ imply $\Gamma, \Delta \rhd s : \sigma$.

Note that the γ-typability can be assured by getting a solution of the typing $\Gamma, \Delta, f : \sigma \rhd s : \sigma$. A *type assertion* can be regarded as such a process. The δ-typability seems to be harder to assure, but it is almost trivial for first order functions like arithmetic operations. And for higher order functions, it may be difficult though only one fixed proof is needed.

Finally, we can prove the type preserving property of reductions, in other words *the subject reduction property.*

Lemma 4.3.10 (subject reduction) If $\Gamma, \Delta \rhd s : \sigma$ (a) and $s \longrightarrow s'$ then $\Gamma, \Delta \rhd s' : \sigma$.

Proof If (a) is obtained by (SUB), (INT) or (GEN), we get this easily from the induction hypothesis of the proof tree, since they are independent from s. So we assume that (a) was not one of them, and proceed to structural induction on s.

Case $s = f \in \Sigma_U$ and $f \to s' \in \gamma$. It is assured by the γ-typability.

Case $s = tu$. The only candidate left for (a) is (APP). So we have $\Gamma, \Delta \rhd t : \rho \to \sigma$ (b) and $\Gamma, \Delta \rhd u : \rho$ for some ρ.

Case $t \longrightarrow t'$ and $s' = t'u$. By the induction hypothesis we have $\Gamma, \Delta \rhd t' : \rho \to \sigma$. Applying (APP) we get our goal.

Case $u \longrightarrow u'$ and $s' = tu'$. Analogous to above.

Case $t = f \in \Sigma_P$ and $fu \longrightarrow^{\varepsilon} s'$. It is assured by the δ-typability.

Case $t = \lambda z.\, s''$ and $s' = s''[z \mapsto u]$. Applying Lemma 4.3.7 to (b) we get $\Gamma, \Delta, z : \rho \rhd s'' : \sigma$. Thus together with $\Gamma, \Delta \rhd u : \rho$, Lemma 4.3.3 can be applied and we get $\Gamma, \Delta \rhd s''[z \mapsto u] : \sigma$.

Case $s = (t, u)$. The only candidate for (a) is (PAIR), thus we have $\sigma = \tau \times \rho$ with $\Gamma, \Delta \rhd t : \tau$ and $\Gamma, \Delta \rhd u : \rho$.

Case $t \longrightarrow t'$ and $s' = (t', u)$, by induction hypothesis we have $\Gamma, \Delta \rhd t' : \tau$. By (PAIR) we get our goal.

Case $u \longrightarrow u'$ and $s' = (t, u')$ is analogous.

Case $s = \lambda y.\, u$, $s' = \lambda y.\, u'$ and $u \longrightarrow u'$. We show by induction on (a).

Case (BASE), (ERR), (ASSUMP), (APP) or (PAIR). They are not applicable.

Case (ABST). We have $\sigma = \tau \to \rho$ and $\Gamma, \Delta, y : \tau \rhd u : \rho$. By the induction hypothesis we get $\Gamma, \Delta, y : \tau \rhd u' : \rho$. Applying (ABST) we get $\Gamma, \Delta \rhd \lambda y.\, u' : \tau \to \rho$.

Case (SUB), (INST) or (GEN). They are already considered.

□

4.4 Consequent soundness theorems

As consequences of the subject reduction property, we immediately get following useful theorems.

Theorem 4.4.1 (strong soundness) $\Gamma, \Delta \rhd s : \sigma$ and $s \xrightarrow{*} s'$ imply $\Gamma, \Delta \rhd s' : \sigma$.

Proof Obviously shown by Lemma 4.3.10 and induction on $s \xrightarrow{n} s'$. □

Theorem 4.4.2 (weak soundness) Let σ be a type such that $\Gamma, \Delta \rhd \mathsf{error} : \sigma$ does not hold. If $\Gamma, \Delta \rhd s : \sigma$ then $s \xrightarrow{*} s'$ implies $s' \neq \mathsf{error}$.

Proof Obvious. □

In order to detect a type error with the error type E, of course we want no value but error be typed by E. The following assumption can assure this.

Assumption 4.4.3 (the error type) We assume $\Gamma(f) \neq \mathsf{E}$ for any pre-defined function f.

Theorem 4.4.4 (error soundness) Let $s \in \Lambda(\emptyset)$. If $\Gamma, \Delta \rhd s : \mathsf{E}$ then s has no normal form but error.

Proof Let v be a normal form of s. By the strong soundness we have $\Gamma, \Delta \triangleright v : \mathsf{E}$.

Case (BASE), (ABST), (PAIR), (GEN) or (INT). They are not applicable because of the structure of E.

Case (ERR) is the trivial case.

Case (ASSUMP). Because v is not a variable or user-defined symbol, Assumption 4.4.3 denies this case.

Case (APP). We have $v = v'v''$, but Lemma 3.1.9 shows that this is not possible.

Case (SUB). By the assumption for the error type, only (REF) can be used. Thus it is trivial from the induction hypothesis.

□

4.5 Totality of types

Definition 4.5.1 For a type constraint Γ, a type σ is n**th-level total** iff $\sigma \in \mathcal{T}_n$ which is defined inductively as follows:

1. $\mathcal{T}_0 := \mathcal{T}$
2. $\sigma \in \mathcal{T}_{n+1}$ if
   (a) $\forall \tau \in \mathcal{T}_n.\ \exists \rho \in \mathcal{T}_n.\ \Gamma \triangleright \tau \to \rho \subseteq \sigma$ and
   (b) $\Gamma \triangleright \sigma_1 \times \sigma_2 \subseteq \sigma \implies \exists \sigma_1', \sigma_2' \in \mathcal{T}_n.\ \Gamma \triangleright \sigma_1' \times \sigma_2' \subseteq \sigma$

We say σ is **total** iff $\sigma \in \mathcal{T}_\omega$, which is defined by $\mathcal{T}_\omega := \bigcap_{n \geq 0} \mathcal{T}_n$.

For example, E is total because it is 0th-level total and for every τ we have $\Gamma \triangleright \tau \to \mathsf{E} \subseteq \mathsf{E}$. Thus base types are also total because $\Gamma \triangleright \tau \to \mathsf{E} \subseteq \iota$ for every τ. For the same reason, products of total types are also total because they satisfy (b).

Note that restriction (b) allows the type $\sigma = (\pi\alpha\beta.\,\alpha \times \beta \to \alpha) \cap (\pi\alpha\beta.\,\alpha \times \beta)^{\mathsf{C}} \to \mathsf{E}$ to be total. Without (b), $\tau_1 \times \tau_2$ become total for any non-total τ_1 and τ_2, thus any ρ such that $\Gamma \rhd \tau_1 \times \tau_2 \to \rho \subseteq \sigma$ cannot satisfy (a).

Theorem 4.5.2 Let $X = \mathcal{X} \cap \mathsf{Dom}(\Delta)$.

If $\mathsf{Ran}(\Delta) \subseteq \mathcal{T}_\omega$, then for every $s \in \Lambda(X)$ there exist $\sigma \in \mathcal{T}_\omega$ with $\Gamma, \Delta \rhd s : \sigma$.

Proof By structural induction on s.

Case $s \in \Sigma \cup X$. Obvious from the assumption.

Case $s = tu$. By the induction hypothesis we have $\tau, \rho \in \mathcal{T}_\omega$ with $\Gamma, \Delta \rhd t : \tau$ and $\Gamma, \Delta \rhd u : \rho$. By the totality of τ, we have $\sigma \in \mathcal{T}_\omega$ and the following proof:

$$\dfrac{\dfrac{\dfrac{I.H.}{\Gamma, \Delta \rhd t : \tau} \quad \dfrac{I.H.}{\Gamma \rhd \rho \to \sigma \subseteq \tau}\,(\text{SUB})}{\Gamma, \Delta \rhd t : \rho \to \sigma} \quad \dfrac{I.H.}{\Gamma, \Delta \rhd u : \rho}}{\Gamma, \Delta \rhd tu : \sigma}\,(\text{APP})$$

Case $s = \lambda y.\,u$. By the induction hypothesis, for any $\tau \in \mathcal{T}_\omega$ we have $\rho \in \mathcal{T}_\omega$ with $\Gamma, \Delta, y : \tau \rhd u : \rho$. Applying (ABST) we get $\Gamma, \Delta \rhd \lambda y.\,u : \tau \to \omega$. We may choose for example $\pi\alpha.\,\alpha$ as τ. Then we see $\tau' \subseteq \tau$ for every τ', and so $\sigma \in \mathcal{T}_\omega$.

Case $s = (t, u)$. By the induction hypothesis we have $\tau, \rho \in \mathcal{T}_\omega$ with $\Gamma, \Delta \rhd t : \tau$ and $\Gamma, \Delta \rhd u : \rho$. Applying (PAIR) we get $\Gamma, \Delta \rhd (t, u) : \tau \times \tau$, and this type is total.

□

Chapter 5

Examples

We give $\cap$ an alias $\mid$ which binds weaker than $\to$. We abbreviate $\sigma \cap \tau^{\mathsf{c}}$ by $\sigma \setminus \tau$.

5.1 An arithmetic example

Example 5.1.1 (factorial) Let base types $\mathcal{B} = \{\mathsf{n}, \mathsf{z}, \mathsf{p}, \mathsf{i}, \mathsf{t}, \mathsf{f}, \mathsf{b}\}$ with

$$\begin{aligned}
\Sigma_{\mathsf{n}} &= \{n \in \mathbb{Z} \mid n < 0\} \\
\Sigma_{\mathsf{z}} &= \{0\} \\
\Sigma_{\mathsf{p}} &= \{n \in \mathbb{Z} \mid 0 < n\} \\
\Sigma_{\mathsf{i}} &= \mathbb{Z} \\
\Sigma_{\mathsf{t}} &= \{\texttt{true}\} \\
\Sigma_{\texttt{false}} &= \{\texttt{false}\} \\
\Sigma_{\mathsf{b}} &= \{\texttt{true}, \texttt{false}\}
\end{aligned}$$

and pre-defined functions $\Sigma_P = \{\texttt{if}, \texttt{zero?}, *, \texttt{pred}\}$ with

$$\delta = \begin{cases}
\texttt{if}(\texttt{true}, x, y) & \to x & \\
\texttt{if}(\texttt{false}, x, y) & \to y & \\
\texttt{zero?}\ 0 & \to \texttt{true} & \\
\texttt{zero?}\ n & \to \texttt{false} & \text{for } n \in \mathbb{Z} \setminus \{0\} \\
m * n & \to m \times n & \text{for } m, n \in \mathbb{Z} \\
\texttt{pred}\ n & \to n - 1 & \text{for } n \in \mathbb{Z}
\end{cases}$$

Let us consider the user defined function $\texttt{fact} \in \Sigma_U$ with the following definition calculates the factorial:

$$\texttt{fact} \to s \in \gamma \quad \text{where } s = \lambda x.\, \texttt{if}(\texttt{zero?}\ x, 1, x * \texttt{fact}(\texttt{pred}\ x))$$

We can see following Δ satisfies δ-typability:

$$\begin{aligned}
\Delta(\texttt{if}) &= \pi\alpha\beta.\, \mathsf{t} \times \alpha \times \beta \to \alpha \mid \mathsf{f} \times \alpha \times \beta \to \alpha \mid \mathsf{b}^{\mathsf{C}} \times \alpha \times \beta \to \mathsf{E} \\
\Delta(\texttt{zero?}) &= \mathsf{z} \to \mathsf{t} \mid \mathsf{n} \cup \mathsf{p} \to \mathsf{f} \mid \mathsf{i}^{\mathsf{C}} \to \mathsf{E} \\
\Delta(*) &= \mathsf{p} \times \mathsf{p} \to \mathsf{p} \mid \mathsf{i} \times \mathsf{i} \to \mathsf{i} \mid (\mathsf{i} \times \mathsf{i})^{\mathsf{C}} \to \mathsf{E} \\
\Delta(\texttt{pred}) &= \mathsf{n} \cup \mathsf{z} \to \mathsf{n} \mid \mathsf{p} \to \mathsf{z} \cup \mathsf{p} \mid \mathsf{i}^{\mathsf{C}} \to \mathsf{E} \\
\Delta(\texttt{fact}) &= \mathsf{z} \to \mathsf{p} \mid \mathsf{p} \to \mathsf{p} \mid \mathsf{n} \cup \mathsf{i}^{\mathsf{C}} \to \mathsf{E}
\end{aligned}$$

and also satisfies γ-typability as following proof tree shows.

$$\dfrac{\dfrac{(1)}{\Delta \rhd s : \mathsf{z} \to \mathsf{p}} \quad \dfrac{(2)}{\Delta \rhd s : \mathsf{p} \to \mathsf{p}} \quad \dfrac{(3)}{\Delta \rhd s : \mathsf{n} \cup \mathsf{i}^{\mathsf{C}} \to \mathsf{E}}}{\Delta \rhd s : \mathsf{z} \to \mathsf{p} \mid \mathsf{p} \to \mathsf{p} \mid (\mathsf{z} \cup \mathsf{p})^{\mathsf{C}} \to \mathsf{E}}\ (\text{INT})$$

(1)

$$\dfrac{\dfrac{\dfrac{\vdots}{\cdot \rhd \texttt{if} : \mathsf{t} \times \mathsf{p} \times \mathsf{i} \to \mathsf{p}} \quad \dfrac{\dfrac{\dfrac{\vdots}{\cdot \rhd \texttt{zero?} : \mathsf{z} \to \mathsf{t}} \quad \cdot \rhd x : \mathsf{z}}{\cdot \rhd \texttt{zero?}\ x : \mathsf{t}} \quad \cdot \rhd 1 : \mathsf{p} \quad \dfrac{\vdots}{\cdot \rhd \cdots : \mathsf{i}}}{\cdot \rhd (\texttt{zero?}\ x, 1, \cdots) : \mathsf{t} \times \mathsf{p} \times \mathsf{i}}\ (\text{APP})}{\Delta, x : \mathsf{z} \rhd \texttt{if}(\texttt{zero?}\ x, 1, \cdots) : \mathsf{p}}\ (\text{ABST})}{\Delta \rhd s : \mathsf{z} \to \mathsf{p}}$$

(2)

$$\dfrac{\dfrac{\dfrac{\vdots}{\cdot \rhd \texttt{if} : \mathsf{f} \times \mathsf{p} \times \mathsf{p} \to \mathsf{p}} \quad \dfrac{\dfrac{\dfrac{\vdots}{\cdot \rhd \texttt{fact} : \mathsf{p} \cup \mathsf{z} \to \mathsf{p}} \quad \dfrac{\vdots}{\cdot \rhd \texttt{pred}\ x : \mathsf{p} \cup \mathsf{z}}}{\cdot \rhd \texttt{fact}(\texttt{pred}\ x) : \mathsf{p}}}{\cdot \rhd (\texttt{zero?}\ x, 1, \texttt{fact}(\texttt{pred}\ x)) : \mathsf{f} \times \mathsf{p} \times \mathsf{p}}\ (\text{APP})}{\Delta, x : \mathsf{p} \rhd \texttt{if}(\texttt{zero?}\ x, 1, \texttt{fact}(\texttt{pred}\ x)) : \mathsf{p}}\ (\text{ABST})}{\Delta \rhd s : \mathsf{p} \to \mathsf{p}}$$

(3)

$$\dfrac{\dfrac{\vdots}{\cdot \triangleright \mathtt{if} : \mathsf{E} \cup \mathsf{f} \times \mathsf{p} \times \mathsf{E} \to \mathsf{E}} \qquad \dfrac{\dfrac{\vdots}{\cdot \triangleright \mathtt{fact} : \mathsf{E} \cup \mathsf{n} \to \mathsf{E}} \qquad \dfrac{\dfrac{\vdots}{\cdot \triangleright \mathtt{pred} : \mathsf{n} \cup \mathsf{i}^{\mathsf{C}} \to \mathsf{E} \cup \mathsf{n}}}{\cdot \triangleright (\mathtt{pred}\ x) : \mathsf{E} \cup \mathsf{n}}}{(\mathtt{zero?}\ x, 1, \mathtt{fact}(\mathtt{pred}\ x) : \mathsf{E} \cup \mathsf{f} \times \mathsf{p} \times \mathsf{E}}}{\dfrac{\Delta, x : \mathsf{n} \cup \mathsf{i}^{\mathsf{C}} \triangleright \mathtt{if}(\mathtt{zero?}\ x, 1, \cdots) : \mathsf{E}}{\Delta \triangleright s : \mathsf{n} \cup \mathsf{i}^{\mathsf{C}} \to \mathsf{E}}}$$

5.2 Type of error handlers

Example 5.2.1 Let $\mathtt{try} \in \Sigma_P$ and

$$\begin{aligned} \mathtt{try}(\mathsf{error}, h) &\to h \quad \in \quad \delta \\ \mathtt{try}(v, h) &\to v \quad \in \quad \delta \quad \text{for } v \in \Lambda_V \setminus \{\mathsf{error}\} \end{aligned}$$

Thanks to the error type, even `try` can be typed as follows:

$$\mathtt{try} : \pi\alpha\beta.\, \mathsf{E} \times \beta \to \beta \mid \alpha \cap \mathsf{E}^{\mathsf{C}} \times \beta \to \alpha$$

This typing can be easily extended for more general error handling. Let us consider a system where an error is expressed by the symbol error paired with its error code n.

$$\begin{aligned} \mathtt{try}'((\mathsf{error}, n), h) &\to h\, n \quad \in \quad \delta \\ \mathtt{try}'(v, h) &\to v \quad \in \quad \delta \quad \text{for } v \in \Lambda_V \setminus \{(\mathsf{error}, n) \mid n \in \Lambda\} \end{aligned}$$

Such $\mathtt{try}'$ can be typed as follows:

$$\mathtt{try}' : \pi\alpha\beta\gamma.(\mathsf{E} \times \beta) \times (\beta \to \gamma) \to \gamma \mid \alpha \cap \mathsf{E}^{\mathsf{C}} \times (\beta \to \gamma) \to \alpha$$

5.3 Expressing evaluation strategies

Example 5.3.1 (call-by-value if) Let $\mathtt{cbv\text{-}if} \in \Sigma_P$ with

$$\mathtt{cbv\text{-}if}(\mathtt{false}, v, y) \to y, \mathtt{cbv\text{-}if}(\mathtt{true}, x, v) \to \mathtt{true} \in \delta \quad \text{iff } v \in \Lambda_V \setminus \mathsf{E}$$

We can give `cbv-if` a type which is different to that of `if`:

$$\begin{aligned}\texttt{cbv-if}: \quad \pi\alpha\beta.\ & \mathsf{t} \times (\alpha \setminus \mathsf{E}) \times \beta \to \alpha \\ \mid\ & \mathsf{t} \times \mathsf{E} \times \beta \to \mathsf{E} \\ \mid\ & \mathsf{f} \times \alpha \times (\beta \setminus \mathsf{E}) \to \beta \\ \mid\ & \mathsf{f} \times \alpha \times \mathsf{E} \to \mathsf{E} \\ \mid\ & \mathsf{b}^{\mathsf{C}} \times \alpha \times \beta \to \mathsf{E}\end{aligned}$$

where the call-by-value strategy of `cbv-if` is concerned. For instance, we have $\texttt{cbv-if}(\texttt{true}, 1, \mathsf{error}) : \mathsf{error}$, while $\texttt{if}(\texttt{true}, 1, \mathsf{error}) : \mathsf{i}$.

Chapter 6

Conclusions

6.1 Concludions

In this thesis we presented an extension of soft typing. This extension gives a program one the following three kinds of types:

- σ such that E is not a subtype of σ. In this case it is assured that the program does not raise an error. So in this sense this typing is sound.
- E itself. In this case it is assured that the program always raise a runtime error, at least if it terminates. So in this sense this typing is complete.
- σ such that E is a subtype of σ. In this case it is not known if the program raises an error. We may insert explicit runtime checks there, so in this sense this typing is soft.

The soundness was proved in a syntactic method. This approach gave us a small profit that we did not need a syntactic restriction for types such as the well-formedness of ideal model.

6.2 Futher works

Our future goal is designing a type assignment algorithm. One of the most difficulties in this work will be treatment of the (SUB) rule. As we have seen

in Section 4.3, (SUB) rule is not needed for (ABST) and (PAIR) rules. Though we need it for (APP) rule, we are expecting that (APP) and (SUB) rules can be welded:

$$\frac{\Gamma, \Delta \rhd s : \sigma \quad \Gamma, \Delta \rhd t : \tau \quad \Gamma \rhd \tau \to \rho \subseteq \sigma}{st : \rho} \text{ (APPSUB)}$$

The type assignment algorithm we want should assign a total type to every expression, in order to assure further assignment will not fail. This may require (INT) to follow (ABST). Thus we need an appropriate strategy for application of (INT) and (ABST).

Acknowledgement

I would like to thank my principal advisor Associate Professor Keiichirou Kusakari for his great help. And I thank Professor Masahiro Sakai for his assist and discussion on this study. Also I am grateful to Professor Toshiki Sakabe, and Assistant Professor Naoki Nishida for their kind guidance.

Bibliography

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[2] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[3] F. Baarder and T. Nipkow. *Term Rewriting and All That*. Cambridge Univ. Press, 1998.

[4] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[5] C. A. Gunter. The semantics of types in programming languages. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 395–475. Clarendon Press, 1994.

[6] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *Information and Control*, volume 71, pages 95–130, 1986.

[7] A. R. Meyer. What is a model of the lambda calculus? In *Information and Control*, volume 52, pages 87–122, 1982.

[8] R. Milner. A theory of type polymorphism in programming. In *Journal of Computer and System Sciences*, pages 348–375, 1978.

[9] F. Müller. Confluence of the lambda calculus with left-linear algebraic rewriting. *Information Processing Letters*, 41(6):293–299, 1992.

[10] M. Widera. An algorithm for checking the disjointness of types. In *2nd Workshop on Scheme and Functional Programming*, pages 65–73, September 2001.

[11] M. Widera. A sketch of complete type inference for functional programming. In *International Workshop on Functional and (Constraint) Logic Programming (WLFP 2001)*, September 2001.

[12] A. K. Wright and R. Cartwright. A practical soft type system for scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.

[13] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1992.