社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2007-58(2008-3)

動的型言語への柔らかい型付けによるエラー検出

山田 晃久† 草刈圭一朗† 酒井 正彦† 坂部 俊樹† 西田 直樹†

† 名古屋大学大学院 情報科学研究科 〒 464-8603 名古屋市千種区不老町

あらまし LISP のような動的型言語のための柔らかい型の表現力豊かな拡張を提案し，その健全性を示す．この拡張では特に、常に型エラーを起こす式を表す「エラー型」と言う特殊な型を導入する．エラー型を含まない型を持つ式では、実行時の型チェックを省くことが出来る．逆にエラー型そのものが付く式は必ずエラーを引き起こすことがわかるので，このような式はコンパイル時に排除することが出来る．

キーワード 動的型言語，柔らかい型，完全な型

Error Detection with Soft Typing for Dynamically Typed Languages

Akihisa YAMADA†, Keiichirou KUSAKARI†, Masahiko SAKAI†,

Toshiki SAKABE†, and Naoki NISHIDA†

† Nagoya University Graduate School of Information Science
Furo-cho, Chikusa-ku, Nagoya, 464-8603, Japan

Abstract We propose a rich extension of soft typing for dynamically-typed languages like LISP, and prove its soundness. This extension, for example, introduces a special type called "error-type", which denotes a type of such an expression whose evaluation always causes a type error. If an expression is typed without using the "error-type", it means that no runtime check is necessary. Conversely, "error-typed" expressions should be rejected at compile-time, since their evaluation always causes an error.

Key words dynamically typed language, soft typing, complete typing

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. Though, it is known that they are not *complete*, i.e. some meaningful programs are rejected.

In a dynamically typed language like LISP, types are checked during program execution. Hence every meaningful programs can be executed. The most important advantage over statically typed languages is its rich expressiveness. Programs are simpler, and more reusable in many context, provided they do not raise a runtime error for the given input.

On the other hand, one of the disadvantages is the difficulty in debugging; programmers must by themselves assure that programs do not raise runtime errors for their intended inputs. In addition, execution is not efficient because of the runtime checks.

Soft typing [3] is a compromise between static and dynamic typing. Like static typing, types are checked at compile time, and codes for runtime checks are omitted for typed programs. Unlike static typing, on the other hand, 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. However, it is not enough for debugging purpose, since programmers still have to assure that the inserted runtime check will not fail.

Complete typing [9] is a concept of typing which detects inevitable type errors. With a complete type system, debugging become easier because programmers can immediately see a 'completely' untyped program have a bug. However, a complete type system needs to decide *type disjointness*, whose efficient algorithm is not yet obtained [8].

In this paper, we extend soft typing to simulate "complete typing". The principal idea 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, 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:

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

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

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

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

In this paper, 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* [5]. Instead we stay syntactic and imitating [11], 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.

2. Expressions

Let $v \in \mathcal{X}$ be a *variable symbol* (a variable, in short) and $f \in \Sigma$ be a *function symbol* (a function, in short). An *expression* $s \in \Lambda$ is defined by $s ::= x \mid f \mid ss \mid (s,s) \mid \lambda x.\, s$.

The set $\mathsf{FV}(s)$ of *free variables* and $\mathsf{BV}(s)$ of *bound variables* in s are defined as usual, and renaming of bound variables (α-convergence) are considered equal. For $X \subseteq \mathcal{X}$, the set of expressions whose free variables are in X is denoted by $\Lambda(X) \stackrel{\text{def}}{=} \{s \in \Lambda \mid \mathsf{FV}(s) \subseteq X\}$. A (syntactic) *value* $v \in \Lambda_V$ is defined as follows:

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

A *substitution* θ is a mapping $\mathcal{X} \to \Lambda$, which is naturally extended on Λ. We use postfix notation for substitutions. $[x \mapsto s]$ denotes the substitution θ such that $x\theta = s$ and $y\theta = y$ for $y \neq x$. An expression $C[\,] \in \Lambda$ is a *context* iff it contains a special function symbol $\Box$ at exactly one place. $C[s]$ denotes the expression obtained from $C[\,]$ by replacing $\Box$ with s.

In the latter, f ranges over Σ, x, y, z range over $\mathcal{X}$, s, t, u range over Λ, θ ranges over substitutions and $C[\,]$ ranges over contexts.

A pair $l \to r \in \Lambda^2$ is a *rewrite rule* iff $\mathsf{FV}(l) \subseteq \mathsf{FV}(r)$. A *rule set* R is a set of rewrite rules. The *rewrite relation at root position induced by* R is defined by $\xrightarrow[R]{}^{\epsilon} \stackrel{\text{def}}{=} \{\langle l\theta, r\theta\rangle \mid l \to r \in R\}$. The *rewrite relation induced by* R is defined by $\xrightarrow[R]{} \stackrel{\text{def}}{=} \{\langle C[s], C[t]\rangle \mid s \xrightarrow[R]{}^{\epsilon} t\}$. The reflexive transitive closure of $\xrightarrow[R]{}$ is denoted by $\xrightarrow[R]{*}$. An expression s is *normal form* w.r.t. R iff there is no s' s.t. $s \xrightarrow[R]{} s'$. The set of all normal forms are denoted by $\mathsf{NF}(R)$.

Here we define several rewrite relations.

Definition 2.1 The β-*reduction* is the rewrite relation $\xrightarrow[\beta]{}$ induced by the following rule set β:

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

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

Definition 2.2 Let $\Sigma_B \subseteq \Sigma$ 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) $(p,q) \in \Lambda_T$ if $p, q \in \Lambda_T$

Assumption 2.3 (pre-defined functions) Let $\Sigma_P \subseteq \Sigma$ 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 $l \to r \in \delta$, $l = fp$ 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 $\mathtt{succ}\ n \longrightarrow (\mathtt{succ}', n)$.

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

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

Definition 2.4 Let $\mathsf{error} \in \Sigma$ be a new constant denoting a type error. The following rule set ϵ defines when a type error occurs:

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

Lemma 2.5 Let s be a closed expression such that $s \in \mathsf{NF}(\beta \cup \delta \cup \epsilon)$. 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 necessary.

Finally, we add *user-defined* functions:

Assumption 2.6 Let $\Sigma_U \subseteq \Sigma$ 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 2.5 also holds for $\longrightarrow$, since every expression with a user-defined symbol is reducible with γ.

3. Types

3.1 Syntax of types

Let $\alpha \in \mathcal{V}$ be a *type variable* and $\iota \in \mathcal{B}$ be a *base type*. A type $\sigma \in \mathcal{T}$ is defined by:

$$\sigma ::= \alpha \mid \iota \mid \mathsf{E} \mid \sigma \times \sigma \mid \sigma \cap \sigma \mid \sigma \cup \sigma \mid \sigma \to \sigma \mid \sigma^{\mathsf{C}} \mid \pi\alpha.\sigma \mid \mu\alpha.\sigma$$

Associativities are right for $\times$ and $\to$, and left for $\cap$ and $\cup$.

Bound and free variables, substitutions and contexts are defined analogously to expressions, and renaming of bound variables are considered equal. In the latter ι ranges over base types, $\alpha, \beta, \gamma, \delta$ range over type variables, $\sigma, \tau, \rho, \upsilon$ range over types.

3.2 Subtyping

Definition 3.1 A *subtyping* is a pair of types written $\sigma \subseteq \tau$. A *type constraint* is a subtyping whose one side is a type variable. A *constraint set* Γ is a set of type constraints. Free type variables of Γ is defined by the following:

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

For a type constraint $\sigma \subseteq \tau$, 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 3.2 (values of base types) For each base type ι, we assume the set $\Sigma_\iota \subseteq \Sigma_B$ of basic constants is given. $\Sigma_{\iota^{\mathsf{C}}}$ denotes the set $\Sigma_\iota^{\mathsf{C}} \stackrel{\text{def}}{=} \Lambda_V \setminus \Sigma_\iota$.

Definition 3.3 (subtyping rules) A subtyping $\sigma \subseteq \tau$ (syntactically) *holds* under a constraint set Γ iff $\Gamma \rhd \sigma \subseteq \tau$ has a finite proof using the following inference rules:

$$\text{BASE: } \Gamma \rhd a \subseteq b \quad \text{if } a, b \text{ in form } \iota \text{ or } \iota^{\mathsf{C}} \text{ and } \Sigma_a \subseteq \Sigma_b$$

$$\text{REF: } \Gamma \rhd \sigma \subseteq \sigma \qquad \text{ERR: } \Gamma \rhd \iota \subseteq \mathsf{E}^{\mathsf{C}}$$

$$\text{L-VAR: } \frac{\Gamma, \alpha \subseteq \sigma \rhd \sigma \subseteq \tau}{\Gamma, \alpha \subseteq \sigma \rhd \alpha \subseteq \tau} \qquad \text{R-VAR: } \frac{\Gamma, \tau \subseteq \beta \rhd \sigma \subseteq \tau}{\Gamma, \tau \subseteq \beta \rhd \sigma \subseteq \beta}$$

$$\text{COMP: } \frac{\Gamma \rhd \rho \subseteq \tau \quad \Gamma \rhd \rho^{\mathsf{C}} \subseteq \tau}{\Gamma \rhd \sigma \subseteq \tau}$$

$$\text{LI1: } \frac{\Gamma \rhd \sigma_1 \subseteq \tau}{\Gamma \rhd \sigma_1 \cap \sigma_2 \subseteq \tau} \qquad \text{LI2: } \frac{\Gamma \rhd \sigma_2 \subseteq \tau}{\Gamma \rhd \sigma_1 \cap \sigma_2 \subseteq \tau}$$

$$\text{RI: } \frac{\Gamma \rhd \sigma \subseteq \tau_1 \quad \Gamma \rhd \sigma \subseteq \tau_2}{\Gamma \rhd \sigma \subseteq \tau_1 \cap \tau_2}$$

$$\text{CI: } \frac{\Gamma \rhd \sigma \subseteq \tau_1^{\mathsf{C}} \cup \tau_2^{\mathsf{C}}}{\Gamma \rhd \sigma \subseteq (\tau_1 \cap \tau_2)^{\mathsf{C}}}$$

$$\text{LU: } \frac{\Gamma \rhd \sigma_1 \subseteq \tau \quad \Gamma \rhd \sigma_2 \subseteq \tau}{\Gamma \rhd \sigma_1 \cup \sigma_2 \subseteq \tau}$$

$$\text{RU1: } \frac{\Gamma \rhd \sigma \subseteq \tau_1}{\Gamma \rhd \sigma \subseteq \tau_1 \cup \tau_2} \qquad \text{RU2: } \frac{\Gamma \rhd \sigma \subseteq \tau_2}{\Gamma \rhd \sigma \subseteq \tau_1 \cup \tau_2}$$

$$\text{CU: } \frac{\Gamma \rhd \sigma \subseteq \tau_1^{\mathsf{C}} \cap \tau_2^{\mathsf{C}}}{\Gamma \rhd \sigma \subseteq (\tau_1 \cup \tau_2)^{\mathsf{C}}}$$

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

$$\text{L-REC: } \frac{\Gamma, \alpha \mapsto \sigma, \alpha \subseteq \tau \rhd \sigma \subseteq \tau}{\Gamma \rhd \mu\alpha.\sigma \subseteq \tau} \quad \text{if } \alpha \notin \mathsf{FV}(\tau, \Gamma)$$

$$\text{R-REC: } \frac{\Gamma, \beta \mapsto \tau \rhd \sigma \subseteq \tau}{\Gamma \rhd \sigma \subseteq \mu\beta.\tau} \quad \text{if } \beta \notin \mathsf{FV}(\sigma, \Gamma)$$

$$\text{C-REC: } \frac{\Gamma, \beta \mapsto \tau \rhd \sigma \subseteq \tau^{\mathsf{C}}}{\Gamma \rhd \sigma \subseteq (\mu\beta.\tau)^{\mathsf{C}}} \quad \text{if } \beta \notin \mathsf{FV}(\sigma, \Gamma)$$

$$\text{FUN: } \frac{\Gamma \rhd \tau_1 \subseteq \sigma_1 \quad \Gamma \rhd \sigma_2 \subseteq \tau_2}{\Gamma \rhd \sigma_1 \to \sigma_2 \subseteq \tau_1 \to \tau_2}$$

$$\text{C-FUN: } \frac{\Gamma \rhd \tau_1 \subseteq \sigma_1 \quad \sigma_2 \subseteq \tau_2^{\mathsf{C}}}{\Gamma \rhd \sigma_1 \to \sigma_2 \subseteq (\tau_1 \to \tau_2)^{\mathsf{C}}}$$

$$\text{PROD: } \frac{\Gamma \rhd \sigma_1 \subseteq \tau_1 \quad \Gamma \rhd \sigma_2 \subseteq \tau_2}{\Gamma \rhd \sigma_1 \times \sigma_2 \subseteq \tau_1 \times \tau_2}$$

$$\text{C-P1: } \frac{\Gamma \rhd \sigma_1 \subseteq \tau_1^{\mathsf{C}}}{\Gamma \rhd \sigma_1 \times \sigma_2 \subseteq (\tau_1 \times \tau_2)^{\mathsf{C}}}$$

$$\text{C-P2: } \frac{\Gamma \rhd \sigma_2 \subseteq \tau_2^{\mathsf{C}}}{\Gamma \rhd \sigma_1 \times \sigma_2 \subseteq (\tau_1 \times \tau_2)^{\mathsf{C}}}$$

$$\text{BF: } \frac{\Gamma \rhd \mathsf{E} \subseteq \rho}{\Gamma \rhd \iota \subseteq \tau \to \rho} \qquad \text{PF: } \frac{\Gamma \rhd \mathsf{E} \subseteq \rho}{\Gamma \rhd \sigma_1 \times \sigma_2 \subseteq \tau \to \rho}$$

$$\text{C-FB: } \frac{\Gamma \rhd \sigma_2 \subseteq \mathsf{E}^{\mathsf{C}}}{\Gamma \rhd \sigma_1 \to \sigma_2 \subseteq \iota^{\mathsf{C}}}$$

$$\text{C-FP: } \frac{\Gamma \rhd \sigma_2 \subseteq \mathsf{E}^{\mathsf{C}}}{\Gamma \rhd \sigma_1 \to \sigma_2 \subseteq (\tau_1 \times \tau_2)^{\mathsf{C}}}$$

$$\text{C-PB: } \frac{\Gamma \rhd \sigma \subseteq \pi\alpha\beta.\alpha \times \beta}{\Gamma \rhd \sigma \subseteq \iota^{\mathsf{C}}} \qquad \text{C-BP: } \frac{\Gamma \rhd \sigma \subseteq \iota}{\Gamma \rhd \sigma \subseteq (\tau_1 \times \tau_2)^{\mathsf{C}}}$$

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

$$\text{TRANS: } \frac{\Gamma \rhd \sigma \subseteq \tau \quad \Gamma \rhd \tau \subseteq \rho}{\Gamma \rhd \sigma \subseteq \rho}$$

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

Definition 3.4 A constraint set Γ is *admissible* iff $\sigma \subseteq \alpha, \alpha \subseteq \sigma' \in \Gamma$ implies $\Gamma \rhd \sigma \subseteq \sigma'$.

Lemma 3.5 (transitivity) Let Γ be an admissible constraint set. If $\Gamma \rhd \sigma \subseteq \tau$ and $\Gamma \rhd \tau \subseteq \rho$ then $\Gamma \rhd \sigma \subseteq \rho$.

Proof By induction on the proof trees. □

3.3 Type judgment

Definition 3.6 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* of Γ is defined by $\mathsf{FVRan}(\Delta) \stackrel{\text{def}}{=} \mathsf{FV}(\mathsf{Ran}(\Delta))$.

Definition 3.7 (typing rules) A judgment $s : \sigma$ syntactically holds under an admissible constraint set Γ and a type environment Δ iff $\Gamma, \Delta \rhd s : \sigma$ has a finite proof using the following rules:

$$\text{BASE: } \Gamma, \Delta \rhd c : \iota \quad \text{if } c \in \Sigma_\iota$$

$$\text{ERR: } \Gamma, \Delta \rhd \mathsf{error} : \mathsf{E}$$

$$\text{ASSUMP: } \Gamma, \Delta \rhd x : \Delta(x) \quad \text{if } x \in \mathsf{Dom}(\Delta)$$

$$\text{ABST: } \frac{\Gamma, \Delta, x : \tau \rhd u : \rho}{\Gamma, \Delta \rhd \lambda x.u : \tau \to \rho} \quad \text{if } x \notin \mathsf{Dom}(\Delta)$$

$$\text{APP: } \frac{\Gamma, \Delta \rhd s : \tau \to \sigma \quad \Gamma, \Delta \rhd t : \tau}{\Gamma, \Delta \rhd st : \sigma}$$

$$\text{PAIR: } \frac{\Gamma, \Delta \rhd s : \sigma \quad \Gamma, \Delta \rhd t : \tau}{\Gamma, \Delta \rhd (s, t) : \sigma \times \tau}$$

$$\text{SUB: } \frac{\Gamma, \Delta \rhd s : \sigma \quad \Gamma \rhd \sigma \subseteq \sigma'}{\Gamma, \Delta \rhd s : \sigma'}$$

$$\text{GEN: } \frac{\Gamma, \Delta \rhd s : \sigma}{\Gamma, \Delta \rhd s : \pi\alpha.\sigma} \quad \text{if } \alpha \notin \mathsf{FV}(\Gamma) \cup \mathsf{FVRan}(\Delta)$$

$$\text{INT: } \frac{\Gamma, \Delta \rhd s : \sigma_1 \quad \Gamma, \Delta \rhd s : \sigma_2}{\Gamma, \Delta \rhd s : \sigma_1 \cap \sigma_2}$$

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 3.8 (substitution) Let $y \notin \mathsf{Dom}(\Delta)$. If $\Gamma, \Delta, y : \tau \rhd s : \sigma$ and $\Gamma, \Delta \rhd t : \tau$ then $\Gamma, \Delta \rhd s[y \mapsto t] : \sigma$.

Proof By structural induction. □

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

Lemma 3.9 If $\Gamma, \Delta, y : \tau \rhd s : \sigma$ and $\Gamma \rhd \tau' \subseteq \tau$ then $\Gamma, \Delta, y : \tau' \rhd s : \sigma$.

Proof By induction on structure of $\Gamma, \Delta, y : \tau \rhd s : \sigma$.

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:

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

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 3.10 If $\Gamma \rhd \rho \subseteq \iota$ or $\Gamma \rhd \rho \subseteq \sigma_1 \times \sigma_2$ then $\Gamma, \Delta \rhd \lambda x.\, s : \rho$ cannot hold.

Proof By structural induction. □

With the above lemma, we can show that every function type of an abstraction can be inferred directly by (ABST).

Lemma 3.11 (abstraction) If (a) $\Gamma, \Delta \rhd \lambda x.\, t : \sigma \to \tau$ then $\Gamma, \Delta, x : \sigma \rhd t : \tau$.

Proof By structural 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 (GEN) or (INT). Obvious from the induction hypothesis.

Case (SUB). We have ρ' with $\Gamma, \Delta \rhd \lambda x.\, t : \rho'$ and (b) $\Gamma \rhd \rho' \subseteq \rho$. This case is done by structural induction on (b).

Case (REF). Obvious from the induction hypothesis.

Case (BF) or (PF). We have $\rho = \iota$ or $\rho_1 \times \rho_2$, which are denied by Lemma 3.10.

Case (L-VAR), (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 3.9 we get $\Gamma, \Delta, x : \sigma \rhd t : \tau'$. Applying (SUB) we get our goal.

Case Others are not applicable. □

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

Assumption 3.12 (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 3.13 (δ-typability) For all $f \in \Sigma_P$ and an expression t, we assume that $\Gamma, \Delta \rhd ft : \sigma$ and $ft \xrightarrow[\delta \cup \epsilon]{}^{\epsilon} s$ 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.

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

Lemma 3.14 (subject reduction) If (a) $\Gamma, \Delta \rhd s : \sigma$ 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, since they are independent from s. We prove other cases by 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 (b) $\Gamma, \Delta \rhd t : \rho \to \sigma$ 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 $fu \longrightarrow^{\epsilon} s'$. It is assured by the δ-typability.

Case $t = \lambda z.\, s''$ and $s' = s''[z \mapsto u]$. Applying Lemma 3.11 to (b) we get $\Gamma, \Delta, z : \rho \rhd s'' : \sigma$. Thus together with $\Gamma, \Delta \rhd u : \rho$, Lemma 3.8 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. □

3.4 Soundness theorems

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

Theorem 3.15 (strong soundness) $\Gamma, \Delta \rhd s : \sigma$ and $s \xrightarrow{*} s'$ imply $\Gamma, \Delta \rhd s' : \sigma$.

Proof Obvious from Lemma 3.14. □

Theorem 3.16 (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 from the strong soundness. □

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

Theorem 3.18 (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 $\Gamma, \Delta \rhd v : \mathsf{E}$.

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 3.17 denies this case.

Case (APP). We have $v = v'v''$, but Lemma 2.5 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. Totality of types

In this section, we show that with an appropriate type environment, every expression can be typed.

Definition 4.1 For a type constraint Γ, the set $\mathcal{T}_n$ of *nth-level total* types is defined inductively as follows: $\mathcal{T}_0 \stackrel{\text{def}}{=} \mathcal{T}$ and $\sigma \in \mathcal{T}_{n+1}$ iff

(1) $\forall \tau \in \mathcal{T}_n.\ \exists \rho \in \mathcal{T}_n.\ \Gamma \rhd \tau \to \rho \subseteq \sigma$ and

(2) $\Gamma \rhd \sigma_1 \times \sigma_2 \subseteq \sigma \implies \exists \sigma_1', \sigma_2' \in \mathcal{T}_n.\ \Gamma \rhd \sigma_1' \times \sigma_2' \subseteq \sigma$

The set $\mathcal{T}_\omega$ of *total* types is defined by $\mathcal{T}_\omega \stackrel{\text{def}}{=} \bigcap_{n \geq 0} \mathcal{T}_n$.

For example, E is total because it is 0th-level total and for every τ we have $\Gamma \rhd \tau \to \mathsf{E} \subseteq \mathsf{E}$. Thus base types are also total because $\Gamma \rhd \tau \to \mathsf{E} \subseteq \iota$ for every τ. For the same reason, products of total types are also total because they satisfy (2).

Note that restriction (2) 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 (2), $\tau_1 \times \tau_2$ 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 (1).

Theorem 4.2 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:

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

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 $\Gamma, \Delta \rhd \lambda y.\, u : \tau \to \rho$. We may choose for example $\tau' \cup ({}^{\mathsf{C}}\tau')$ 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. □

5. Examples

We give $\cap$ an alias $|$ which binds weaker than $\to$. We abbreviate $\sigma \cap \tau^{\mathsf{C}}$ by $\sigma - \tau$.

5.1 An arithmetic example

The following example is detail of `fact` mentioned in the introduction.

Example 5.1 (factorial) Let the set of base types $\mathcal{B} = \{\mathsf{negint}, \mathsf{zero}, \mathsf{posint}, \mathsf{int}, \mathsf{true}, \mathsf{false}, \mathsf{bool}\}$ with

$$\Sigma_{\mathsf{true}} = \{\mathtt{true}\}, \Sigma_{\mathsf{false}} = \{\mathtt{false}\},$$

$$\Sigma_{\mathsf{bool}} = \{\mathtt{true}, \mathtt{false}\}, \Sigma_{\mathsf{zero}} = \{0\}, \Sigma_{\mathsf{int}} = \mathbb{Z},$$

$$\Sigma_{\mathsf{negint}} = \{n \in \mathbb{Z} \mid n < 0\} \text{ and } \Sigma_{\mathsf{posint}} = \{n \in \mathbb{Z} \mid 0 < n\}$$

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

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

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

$$\mathtt{fact} \to \lambda x.\, \mathtt{if}(\mathtt{zero?}\ x, 1, x * \mathtt{fact}(\mathtt{pred}\ x)) \in \gamma$$

We can see following Δ satisfies δ-typability:

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

where base types are abbreviated by their capital letters. We can show the following typing satisfies γ-typability:

$$\Delta(\mathtt{fact}) = \mathsf{z} \cup \mathsf{p} \to \mathsf{p} \mid \mathsf{n} \cup \mathsf{i}^{\mathsf{C}} \to \mathsf{E}$$

5.2 Type of error handlers

Example 5.2 Let $\mathtt{try} \in \Sigma_P$ and

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

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

$$\mathtt{try} : \pi\alpha\beta.\, \mathsf{E} \times \beta \to \beta \mid (\alpha - \mathsf{E}) \times \beta \to \alpha$$

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.

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

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

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

5.3 Expressing evaluation strategies

Example 5.3 (call-by-value if) Let $\mathtt{cbv\text{-}if} \in \Sigma_P$ with

$$\left.\begin{array}{l} \mathtt{cbv\text{-}if}(\mathtt{true}, x, v) \to x \\ \mathtt{cbv\text{-}if}(\mathtt{false}, v, y) \to y \end{array}\right\} \in \delta \quad \text{iff } v \in \Lambda_V \setminus \{\mathsf{error}\}$$

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

$$\begin{aligned} \mathtt{cbv\text{-}if} : \pi\alpha\beta.\ & \mathsf{true} \times (\alpha - \mathsf{E}) \times \beta \to \alpha \\ & \mid \mathsf{true} \times \mathsf{E} \times \beta \to \mathsf{E} \\ & \mid \mathsf{false} \times \alpha \times (\beta - \mathsf{E}) \to \beta \\ & \mid \mathsf{false} \times \alpha \times \mathsf{E} \to \mathsf{E} \\ & \mid \mathsf{bool}^{\mathsf{C}} \times \alpha \times \beta \to \mathsf{E} \end{aligned}$$

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

6. Conclusions

In this paper 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; we did not need a syntactic restriction for types such as the well-formedness of ideal model.

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 3.3, (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).

Acknowledgements

This research was partially supported by MEXT. KAKENHI #16650005, #18500011 and #17700009, and by the Kayamori Foundation of Informational Science Advancement.

References

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

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

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

[4] 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.

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

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

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

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

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

[10] 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.

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