IPSJ Transactions on Programming Vol.5 No.2 16–24 (Mar. 2012)

Regular Paper

A Sound Type System for Typing Runtime Errors

Akihisa Yamada[1,a)] Keiichirou Kusakari[1] Toshiki Sakabe[1]
Masahiko Sakai[1] Naoki Nishida[1]

Received: September 30, 2011, Accepted: December 30, 2011

Abstract: Dynamically typed languages such as Scheme are widely adopted because of their rich expressiveness. However, there is the drawback that dynamic typing cannot detect runtime errors at compile time. In this paper, we propose a type system which enables static detection of runtime errors. The key idea of our approach is to introduce a special type, called the error type, for expressions that cause runtime errors. The proposed type system brings out the benefit of the error type with set-theoretic union, intersection and complement types, recursive types, parametric polymorphism and subtyping. While existing type systems usually ensure that evaluation never causes runtime errors for typed expressions, our system ensures that evaluation always causes runtime errors for expressions typed with the error type. Likewise, our system also ensures that evaluation never causes errors for expressions typed with any type that does not contain the error type. Under the usual definition of subtyping, it is difficult to syntactically prove the soundness of our type system. We redefine subtyping by introducing the notion of intransitive subtyping, and syntactically prove the soundness under the new definition.

Keywords: type systems, dynamically typed languages

1. Introduction

In *dynamically typed languages* like Scheme, types are not asserted at compile time. In general, dynamically typed languages are more expressive than statically typed languages; programmers can use functions or variables to range over any values of any types. For example, the following Scheme program defines the function SAT that computes satisfiability of curried boolean functions of any arity:

```
(define (SAT x) (if (boolean? x)
                 x
                 (or (SAT (x #t)) (SAT (x #f)))))
```

A typical input of SAT is

t_1 := (lambda (y) (lambda (z) (or (not y) z)))

SAT works in a straightforward way; if the argument x is a boolean value, then it is satisfiable iff x = #t so x is returned. Otherwise SAT considers x to be a boolean function, and returns true if either (x #t) or (x #f) is satisfiable. The expression (SAT t_1) evaluates to #t because ((t_1 #f) #t) evaluates to #t. Such programming is not allowed in usual statically typed languages, because the variable x does not have a fixed type.

The price for the expressiveness is the difficulty in debugging; erroneous expressions are detected only when their evaluation fails with runtime errors. In the example above, for uncurried version of t_1:

t_2 := (lambda (y z) (or (not y) z))

the expression (SAT t_2) always raises a runtime error, but it will not be rejected by the compiler. Thus, to produce bug-free programs, developers need a careful test scenario to detect all possible runtime errors.

Our work is motivated to help programmers of a dynamically typed language to make bug-free programs by (1) accepting expressions whose evaluation *never* causes runtime errors, (2) rejecting expressions whose evaluation *always* causes runtime errors, and (3) by still retaining the expressiveness of dynamically typed languages. In this paper, we will present a sound type system that provides enough expressive power for this triple goal. We concentrate on the theory, and a type inference algorithm will not be presented here.

The key idea of our work is to allow types to express the situation where runtime errors occur. For this purpose we introduce a special *error type*, denoted by E, representing runtime errors. Using E in type expressions, erroneous situations can be expressed; e.g., $\tau \to \mathsf{E}$ is the type for functions that cause runtime errors for inputs of type τ. The type system assures that (1) evaluation never causes errors for expressions typed with any type that does not contain E (defined later precisely). (2) evaluation always causes runtime errors (or it diverges) for expressions typed with E. To retain the expressiveness of dynamically typed languages, (3) untyped expressions should be evaluated with runtime checks as usual.

To bring out the full benefit of E, our system has a certain expressive power including *intersection type* [7], [20] $\sigma \cap \tau$, set-theoretic *union type* [3], [17] $\sigma \cup \tau$, *complement type* [11], [24] τ^{C}, *universal quantification* [13], [14], [18] $\forall \alpha.\, \tau$, *existential quantification*[*1] $\exists \alpha.\, \tau$, and *recursive type* [17] $\mu\alpha.\, \tau$, together with sub-

1 Graduate School of Information Science, Nagoya University, Nagoya, Aichi 464–8603, Japan
a) ayamada@sakabe.i.is.nagoya-u.ac.jp

*1 We follow MacQueen et al. [17] and quantifications do not have a construct such as pack or open.

typing $\sigma \subseteq \tau$. Using these type constructions, SAT is typed as follows:

$$\texttt{SAT} : (\tau_1 \to \texttt{Bool}) \cap (\tau_1^{\mathsf{C}} \to \mathsf{E})$$

where $\tau_1 = \mu\alpha.\,\texttt{Bool} \cup (\texttt{Bool} \to \alpha)$. This typing means that SAT returns a boolean value for any curried boolean functions and boolean inputs, and always causes a runtime error for other inputs.

In the existing formalization employing semantics for types and/or expressions, it is essential for soundness that erroneous expressions should not have a type. Since we give the new type E for erroneous expressions, a different approach is required to prove the soundness of our system.

The syntactic approach inspired by Wright et al. [26] does not depend on semantics; however, their approach cannot be easily extended for subtyping. Suppose we are trying to prove some predicate P on functional types by induction on the height of the subtyping proof. Transitivity of subtyping allows a deduction such as:

$$\frac{\vdash \sigma \to \tau \subseteq \rho \qquad \vdash \rho \subseteq \sigma' \to \tau'}{\vdash \sigma \to \tau \subseteq \sigma' \to \tau'}\,(\subseteq\text{-TRANS})$$

The induction hypothesis applies for $P(\sigma \to \tau)$, but not for $P(\rho)$ because ρ is not always a functional type. Even worse, ρ may be more complex than $\sigma' \to \tau'$ e.g., $(\sigma \to \tau) \cap \rho$, $(\sigma \to \tau) \cap \rho \cup (\sigma \to \tau)$, $\mu\alpha.(\sigma \to \tau) \cap \rho \cup (\sigma \to \tau)$, and so on.

To overcome this problem, we introduce a non-transitive relation $\trianglelefteq$ called *intransitive subtyping*, and define subtyping $\subseteq$ as the transitive closure of $\trianglelefteq$. Under this definition, we show that syntactically complex subtypes can be ignored for typing values, which is the key to our soundness proof.

The rest of this paper is organized as follows: Section 2 presents our target functional language and Section 3 introduces the type system. Section 4 proves the subject reduction property of the system and Section 5 presents soundness theorems. Section 6 demonstrates how the system works using the SAT example. Section 7 describes related works, and Section 8 gives concluding remarks.

2. Expressions

In this section, we define the functional language we consider.

Definition 1 Let $\mathcal{X}$ be a set of *variables*, C be a set of *constants*, $\mathcal{F}$ be a set of built-in *functions*, and e be a special symbol representing a runtime error. The set Λ of *expressions* and the set $\mathcal{V}al$ of *values* are defined by the following grammar:

$$t \ ::= \ x \mid c \mid f \mid (t,t) \mid t\ t \mid \lambda x.\,t \mid \mathsf{e} \qquad (\Lambda)$$

$$v \ ::= \ c \mid f \mid (v,v) \mid \lambda x.\,t \mid \mathsf{e} \qquad (\mathcal{V}al)$$

We use meta-variables x, y, z for variables, c for a constant, f for a function, r, s, t for expressions, and u, v for values.

The set $\mathsf{FV}(t)$ of *free variables* of t is defined as usual. We assume bound variables are renamed to avoid capture. The *substitution* of x by s in t is defined as usual and written $t[x \mapsto s]$.

Definition 2 The *one-step reduction* relation $\longrightarrow$ is defined in **Fig. 1**. The *reduction* relation $\twoheadrightarrow$ is the reflexive transitive closure of $\longrightarrow$.

$$(\lambda x.\,t)\ v \longrightarrow t[x \mapsto v] \qquad (\beta_{\mathrm{v}})$$

$$f\ v \longrightarrow \delta(f, v) \qquad (\delta)$$

$$c\ v \longrightarrow \mathsf{e} \qquad (u_1, u_2)\ v \longrightarrow \mathsf{e} \qquad \mathsf{e}\ v \longrightarrow \mathsf{e} \qquad (\epsilon)$$

$$\frac{s \longrightarrow s'}{s\ t \longrightarrow s'\ t} \qquad \frac{t \longrightarrow t'}{s\ t \longrightarrow s\ t'} \qquad \frac{s \longrightarrow s'}{(s,t) \longrightarrow (s',t)} \qquad \frac{t \longrightarrow t'}{(s,t) \longrightarrow (s,t')}$$

Fig. 1 Rules for one-step reduction.

The rule (β_{v}) represents the *call-by-value* β-reduction. The rule (δ) reduces applications of built-in functions whose interpretations are given by a total function $\delta : \mathcal{F} \times \mathcal{V}al \to \mathcal{V}al$. Note that δ is total; if f should be interpreted as a partial function which is not defined for input v, then $\delta(f, v)$ should be defined to be the error symbol e. An application of non-function is also reduced to e by (ϵ).

Note that our definition allows values to contain errors. In order to exclude such situations, we introduce the notion of *safe values*:

Definition 3 (Safe Values) The set $\mathcal{V}al_{safe}$ of *safe values* is the least set satisfying:

- $c, f, \lambda x.\,t \in \mathcal{V}al_{safe}$
- $v_1, v_2 \in \mathcal{V}al_{safe} \implies (v_1, v_2) \in \mathcal{V}al_{safe}$

3. The Type System

Definition 4 Let $\mathcal{B}$ be a set of *base types* and $\mathcal{X}_{\mathcal{T}}$ be a set of *type variables*. The set $\mathcal{T}$ of *types* is defined by the following grammar:

$$\tau \ ::= \ \alpha \mid \iota \mid \tau \to \tau \mid \tau \times \tau \mid \tau \cap \tau \mid \tau \cup \tau \mid \tau^{\mathsf{C}} \mid \forall\alpha.\,\tau \mid \exists\alpha.\,\tau \mid \mu\alpha.\,\tau \mid \mathsf{E}$$

We use meta-variables α, β, for type variables, ι for a base type and ρ, σ, τ for types.

Parentheses are added to avoid ambiguity with precedence in order $\circ^{\mathsf{C}}, \times, \cap, \cup, \to$ and binding operators. Notions of free type variables, renaming of bound type variables and type substitution are defined analogously to expressions.

Definition 5 For every base type ι, we assume the set $C_\iota \subseteq C$ is given. A *subtyping environment* Γ is a set of subtyping assumptions written $\alpha_1 \trianglelefteq \alpha'_1, \ \cdots, \ \alpha_n \trianglelefteq \alpha'_n$ for distinct α_i and α'_i. A formula in form $\Gamma \vdash \tau \trianglelefteq \tau'$ is called *intransitive subtyping*, whose validity is defined by the rules in **Fig. 2**. *Subtyping* $\Gamma \vdash \tau \subseteq \tau'$ is defined as the transitive closure of the intransitive subtyping.

We write the type $\exists\alpha.\,\alpha$ by $\top$, and $\forall\alpha.\,\alpha$ by $\bot$. $\top$ is the maximum and $\bot$ is the minimum type w.r.t. subtyping.

Definition 6 For each $f \in \mathcal{F}$, we assume a set $\mathsf{Ty}(f)$ of types in form $\sigma \to \tau$ is given. A *type environment* Δ is a set of assumptions in form $x_1 : \tau_1, \ \cdots, \ x_n : \tau_n$ for distinct x_i. A formula in form $\Delta \vdash t : \tau$ is called a *type judgment*, whose validity is defined by the rules in **Fig. 3**.

The function Ty gives the basis for typing built-in functions. To ensure type preservation under δ-reduction, we assume the following:

Assumption 7 (δ-typability) For all $f \in \mathcal{F}$ and $\sigma \to \tau \in \mathsf{Ty}(f)$, we assume

Basic:

$$\Gamma \vdash \tau \trianglelefteq \tau \quad (\trianglelefteq\text{-REF})$$
$$\Gamma,\ \alpha \trianglelefteq \alpha' \vdash \alpha \trianglelefteq \alpha' \quad (\trianglelefteq\text{-ASS})$$
$$\frac{\Gamma \vdash \sigma' \trianglelefteq \sigma \qquad \Gamma \vdash \tau \trianglelefteq \tau'}{\Gamma \vdash \sigma \to \tau \trianglelefteq \sigma' \to \tau'} \quad (\trianglelefteq\text{-}\to)$$
$$\Gamma \vdash \iota \trianglelefteq \iota' \text{ if } \mathcal{C}_\iota \subseteq \mathcal{C}_{\iota'} \quad (\trianglelefteq\text{-}\mathcal{B})$$
$$\frac{\Gamma \vdash \sigma \trianglelefteq \sigma' \qquad \Gamma \vdash \tau \trianglelefteq \tau'}{\Gamma \vdash \sigma \times \tau \trianglelefteq \sigma' \times \tau'} \quad (\trianglelefteq\text{-}\times)$$

Intersection and Union:

$$\Gamma \vdash \tau \trianglelefteq \tau \cap \tau \quad (\trianglelefteq\text{-}\cap\text{I}) \qquad \Gamma \vdash \tau_1 \trianglelefteq \tau_1 \cup \tau_2 \quad (\trianglelefteq\text{-}\cup\text{I})$$
$$\Gamma \vdash \tau_1 \cap \tau_2 \trianglelefteq \tau_1 \quad (\trianglelefteq\text{-}\cap\text{E}) \qquad \Gamma \vdash \tau \cup \tau \trianglelefteq \tau \quad (\trianglelefteq\text{-}\cup\text{E})$$
$$\frac{\Gamma \vdash \tau_1 \trianglelefteq \tau_1' \qquad \Gamma \vdash \tau_2 \trianglelefteq \tau_2'}{\Gamma \vdash \tau_1 \cap \tau_2 \trianglelefteq \tau_1' \cap \tau_2'} \quad (\trianglelefteq\text{-}\cap\text{C}) \qquad \frac{\Gamma \vdash \tau_1 \trianglelefteq \tau_1' \qquad \Gamma \vdash \tau_2 \trianglelefteq \tau_2'}{\Gamma \vdash \tau_1 \cup \tau_2 \trianglelefteq \tau_1' \cup \tau_2'} \quad (\trianglelefteq\text{-}\cup\text{C})$$

Distribution:

$$\Gamma \vdash \sigma \cap (\tau_1 \cup \tau_2) \trianglelefteq (\sigma \cap \tau_1) \cup (\sigma \cap \tau_2) \quad (\trianglelefteq\text{-}\cap\cup)$$
$$\Gamma \vdash (\sigma \to \tau_1) \cap (\sigma \to \tau_2) \trianglelefteq \sigma \to (\tau_1 \cap \tau_2) \quad (\trianglelefteq\text{-}\to\cap)$$
$$\Gamma \vdash (\sigma_1 \to \tau) \cap (\sigma_2 \to \tau) \trianglelefteq (\sigma_1 \cup \sigma_2) \to \tau \quad (\trianglelefteq\text{-}\cup\to)$$

Recursion:

$$\Gamma \vdash \tau[\alpha \mapsto \mu\alpha.\tau] \trianglelefteq \mu\alpha.\tau \quad (\trianglelefteq\text{-}\mu\text{I}) \qquad \Gamma \vdash \mu\alpha.\tau \trianglelefteq \tau[\alpha \mapsto \mu\alpha.\tau] \quad (\trianglelefteq\text{-}\mu\text{E})$$
$$\frac{\Gamma,\ \alpha \trianglelefteq \alpha' \vdash \tau \trianglelefteq \tau'}{\Gamma \vdash \mu\alpha.\tau \trianglelefteq \mu\alpha'.\tau'} \text{ if } \alpha \notin \mathsf{FV}(\Gamma, \tau') \text{ and } \alpha' \notin \mathsf{FV}(\Gamma, \tau) \quad (\trianglelefteq\text{-}\mu\text{C})$$

Universal and Existential Quantification:

$$\Gamma \vdash \forall\alpha.\tau \trianglelefteq \tau[\alpha \mapsto \sigma] \quad (\trianglelefteq\text{-}\forall\text{E}) \qquad \Gamma \vdash \tau[\alpha \mapsto \sigma] \trianglelefteq \exists\alpha.\tau \quad (\trianglelefteq\text{-}\exists\text{I})$$

Complementation:

$$\Gamma \vdash \iota \trianglelefteq \iota'^{\mathsf{C}} \text{ if } \mathcal{C}_\iota \subseteq \mathcal{C} \setminus \mathcal{C}_{\iota'} \quad (\trianglelefteq\text{-}\mathcal{B}^{\mathsf{C}})$$
$$\Gamma \vdash \sigma \to \tau^{\mathsf{C}} \trianglelefteq (\sigma \to \tau)^{\mathsf{C}} \quad (\trianglelefteq\text{-}\to^{\mathsf{C}})$$
$$\Gamma \vdash \sigma^{\mathsf{C}} \times \tau \trianglelefteq (\sigma \times \tau')^{\mathsf{C}} \qquad \Gamma \vdash \sigma \times \tau^{\mathsf{C}} \trianglelefteq (\sigma' \times \tau)^{\mathsf{C}} \quad (\trianglelefteq\text{-}\times^{\mathsf{C}})$$
$$\Gamma \vdash \iota \trianglelefteq (\sigma \to \tau)^{\mathsf{C}} \quad (\trianglelefteq\text{-}\mathcal{B}{\to}^{\mathsf{C}}) \qquad \Gamma \vdash \iota \trianglelefteq (\sigma \times \tau)^{\mathsf{C}} \quad (\trianglelefteq\text{-}\mathcal{B}{\times}^{\mathsf{C}})$$
$$\Gamma \vdash \sigma \to \tau \trianglelefteq \iota^{\mathsf{C}} \quad (\trianglelefteq\text{-}{\to}\mathcal{B}^{\mathsf{C}}) \qquad \Gamma \vdash \sigma \to \tau \trianglelefteq (\sigma' \times \tau')^{\mathsf{C}} \quad (\trianglelefteq\text{-}{\to}{\times}^{\mathsf{C}})$$
$$\Gamma \vdash \sigma \times \tau \trianglelefteq \iota^{\mathsf{C}} \quad (\trianglelefteq\text{-}{\times}\mathcal{B}^{\mathsf{C}}) \qquad \Gamma \vdash \sigma \times \tau \trianglelefteq (\sigma' \to \tau')^{\mathsf{C}} \quad (\trianglelefteq\text{-}{\times}{\to}^{\mathsf{C}})$$
$$\Gamma \vdash \mathsf{E} \trianglelefteq \iota^{\mathsf{C}} \quad (\trianglelefteq\text{-}\mathsf{E}\mathcal{B}^{\mathsf{C}}) \qquad \Gamma \vdash \mathsf{E} \trianglelefteq (\sigma \to \tau)^{\mathsf{C}} \quad (\trianglelefteq\text{-}\mathsf{E}{\to}^{\mathsf{C}})$$
$$\Gamma \vdash \mathsf{E} \trianglelefteq (\sigma \times \tau)^{\mathsf{C}} \quad (\trianglelefteq\text{-}\mathsf{E}{\times}^{\mathsf{C}})$$
$$\Gamma \vdash \top^{\mathsf{C}} \trianglelefteq \bot \quad (\trianglelefteq\text{-}\top^{\mathsf{C}}) \qquad \Gamma \vdash \top \subseteq \bot^{\mathsf{C}} \quad (\trianglelefteq\text{-}\bot^{\mathsf{C}})$$
$$\Gamma \vdash {\tau_1}^{\mathsf{C}} \cup {\tau_2}^{\mathsf{C}} \trianglelefteq (\tau_1 \cap \tau_2)^{\mathsf{C}} \quad (\trianglelefteq\text{-}\cap^{\mathsf{C}})$$
$$\Gamma \vdash {\tau_1}^{\mathsf{C}} \cap {\tau_2}^{\mathsf{C}} \trianglelefteq (\tau_1 \cup \tau_2)^{\mathsf{C}} \quad (\trianglelefteq\text{-}\cup^{\mathsf{C}})$$
$$\frac{\Gamma \vdash \tau' \trianglelefteq \tau}{\Gamma \vdash \tau^{\mathsf{C}} \trianglelefteq \tau'^{\mathsf{C}}} \quad (\trianglelefteq\text{-}^{\mathsf{C}}\text{C})$$

Subtyping:

$$\frac{\Gamma \vdash \tau \trianglelefteq \tau'}{\Gamma \vdash \tau \subseteq \tau'} \quad (\subseteq\text{-}\trianglelefteq) \qquad \frac{\Gamma \vdash \tau \subseteq \tau' \qquad \Gamma \vdash \tau' \subseteq \tau''}{\Gamma \vdash \tau \subseteq \tau''} \quad (\subseteq\text{-TR})$$

Fig. 2 Subtyping and intransitive subtyping.

$$\Delta,\ x : \tau \vdash x : \tau \quad (\text{ASS}) \qquad \Delta \vdash c : \iota \text{ if } c \in \mathcal{C}_\iota \quad (\text{BASE}) \qquad \Delta \vdash f : \sigma \to \tau \text{ if } \sigma \to \tau \in \mathsf{Ty}(f) \quad (\text{PRIM})$$
$$\frac{\Delta,\ x : \tau \vdash s : \sigma}{\Delta \vdash \lambda x.\, s : \tau \to \sigma} \quad (\to\text{I}) \qquad \frac{\Delta \vdash s : \tau \to \sigma \qquad \Delta \vdash t : \tau}{\Delta \vdash st : \sigma} \quad (\to\text{E}) \qquad \frac{\Delta \vdash t_1 : \tau_1 \qquad \Delta \vdash t_2 : \tau_2}{\Delta \vdash (t_1, t_2) : \tau_1 \times \tau_2} \quad (\times\text{I})$$
$$\frac{\Delta \vdash t : \tau_1 \qquad \Delta \vdash t : \tau_2}{\Delta \vdash t : \tau_1 \cap \tau_2} \quad (\cap\text{I}) \qquad \frac{\Delta,\ x : \sigma_1 \vdash t : \tau \qquad \Delta,\ x : \sigma_2 \vdash t : \tau}{\Delta,\ x : \sigma_1 \cup \sigma_2 \vdash t : \tau} \quad (\cup\text{L}) \qquad \frac{\Delta \vdash t : \tau}{\Delta \vdash t : \forall\alpha.\tau} \text{ if } \alpha \notin \mathsf{FV}(\Delta) \quad (\forall\text{I})$$
$$\frac{\Delta,\ x : \sigma \vdash t : \tau}{\Delta,\ x : \exists\alpha.\sigma \vdash t : \tau} \text{ if } \alpha \notin \mathsf{FV}(\Delta) \cup \mathsf{FV}(\tau) \quad (\exists\text{L}) \qquad \frac{\Delta \vdash t : \tau}{\Delta \vdash t : \tau'} \text{ if } \vdash \tau \trianglelefteq \tau' \quad (\trianglelefteq) \qquad \Delta \vdash \mathsf{e} : \mathsf{E} \quad (\mathsf{E}) \qquad \frac{\Delta \vdash s : (\top \to \top)^{\mathsf{C}} \qquad \Delta \vdash t : \tau}{\Delta \vdash st : \mathsf{E}} \quad (\to^{\mathsf{C}}\text{E})$$

Fig. 3 Rules for type judgment.

$$\Delta \vdash v : \sigma \implies \Delta \vdash \delta(f, v) : \tau.$$

First we show some admissible rules we use freely in the latter discussion.

Proposition 8 The following rules are admissible in our system:

$$\frac{\Delta \vdash t : \tau}{\Delta \vdash t : \tau'} \text{ if } \vdash \tau \subseteq \tau' \quad (\subseteq)$$

$$\frac{\Delta,\ x : \sigma \vdash t : \tau}{\Delta,\ x : \sigma' \vdash t : \tau} \text{ if } \vdash \sigma' \trianglelefteq \sigma \quad (\trianglelefteq\text{L})$$

Proof

($\subseteq$). Obvious by recursively applying ($\trianglelefteq$).

($\trianglelefteq$L). In the deduction of $\Delta,\ x : \sigma \vdash t : \tau$, every occurrence of the axiom

$$\frac{}{\Delta,\ x : \sigma \vdash x : \sigma}\,(\text{ASS})$$

can be replaced by the deduction

$$\frac{\dfrac{}{\Delta,\ x : \sigma' \vdash x : \sigma'}\,(\text{ASS})}{\Delta,\ x : \sigma' \vdash x : \sigma}\,(\trianglelefteq)$$

and we get the proof of $\Delta,\ x : \sigma' \vdash t : \tau$. □

Remark 9 One may wonder why we distinguish $\trianglelefteq$ and $\subseteq$ despite the admissibility of ($\subseteq$). The question is: Is $\vdash \tau \subseteq \tau'$ derived in our system if and only if it is derived in the system where $\trianglelefteq$ and $\subseteq$ are identified? This is not trivial because of the rule ($\trianglelefteq$-μC) where a condition of free type variables must be satisfied in *one-step* of $\trianglelefteq$. Though we expect a positive result, the problem is left open at the time of writing.

Remark 10 The following rule ($\cup$E) is proposed by MacQueen et al. [17]:

$$\frac{\Delta,\ x : \sigma_1 \vdash t : \tau \qquad \Delta,\ x : \sigma_2 \vdash t : \tau \qquad \Delta \vdash s : \sigma_1 \cup \sigma_2}{\Delta \vdash t[x \mapsto s] : \tau} \quad (\cup\text{E})$$

Our system adopts ($\cup$L), which Barbanera [3] showed to be strictly weaker than ($\cup$E). Nonetheless, a restricted form of ($\cup$E), where s is assumed to be a value, can be derived with help of Lemma 13 latter stated. This result is enough to prove type preservation under *call-by-value* β_{v}-reduction. The restriction seems to be reasonable for further extension, e.g., ($\cup$E) is shown to break subject reduction property in *conjunctive-disjunctive* λ-*calculi* [9].

4. Subject Reduction Property

In this section we prove the main property to the soundness of our system, i.e., the *subject reduction*: If $\vdash t : \tau$ and $t \longrightarrow t'$, then $\vdash t' : \tau$.

As mentioned earlier, subtyping becomes a problem in a straightforward approach. The problem is ($\trianglelefteq$), where the premise may be more complex in syntax than the conclusion, because of the rules ($\trianglelefteq$-$\cap$E), ($\trianglelefteq$-$\cup$E), ($\trianglelefteq$-μE) and ($\trianglelefteq$-$\forall$E). By following two lemmas we show that such inversion can be shortcut under the restriction of expressions to values.

Lemma 11 Let $\alpha, \alpha' \notin \mathsf{FV}(\Gamma)$, $\alpha \notin \mathsf{FV}(\tau')$ and $\alpha' \notin \mathsf{FV}(\tau)$. If $\Gamma \vdash \sigma \trianglelefteq \sigma'$ and $\Gamma,\ \alpha \trianglelefteq \alpha' \vdash \tau \trianglelefteq \tau'$, then $\Gamma \vdash \tau[\alpha \mapsto \sigma] \trianglelefteq \tau'[\alpha' \mapsto \sigma']$.

Proof By induction on the height of the proof of $\Gamma,\ \alpha \trianglelefteq \alpha' \vdash \tau \trianglelefteq \tau'$. See Appendix for details. □

Lemma 12 If $\Delta \vdash v : \tau$ has a proof with height h, then there exists a proof shorter than h for:

(1) both $\Delta \vdash v : \tau_1$ and $\Delta \vdash v : \tau_2$ if $\tau = \tau_1 \cap \tau_2$,
(2) either $\Delta \vdash v : \tau_1$ or $\Delta \vdash v : \tau_2$ if $\tau = \tau_1 \cup \tau_2$,
(3) $\Delta \vdash v : \sigma[\alpha \mapsto \mu\alpha.\sigma]$ if $\tau = \mu\alpha.\sigma$,
(4) $\Delta \vdash v : \sigma[\alpha \mapsto \rho]$ if $\tau = \forall\alpha.\sigma$.

Proof By simultaneous induction on the proof of $\Delta \vdash v : \tau$. Lemma 11 is used for statement 3. See Appendix for details. □

The following two lemmas help us to prove type preservation under β_{v}-reduction. Thanks to Lemma 12, both can be proved in a straightforward induction. The detailed proofs are shown in the Appendix.

Lemma 13 (Substitution) Let $x \notin \mathsf{Dom}(\Delta)$. If $\Delta,\ x : \sigma \vdash t : \tau$ and $\Delta \vdash v : \sigma$, then $\Delta \vdash t[x \mapsto v] : \tau$.

Proof By induction on the proof of $\Delta,\ x : \sigma \vdash t : \tau$. □

Lemma 14 (Abstraction) If $\Delta \vdash \lambda x.t : \sigma \to \tau$, then $\Delta,\ x : \sigma \vdash t : \tau$.

Proof By induction on the proof of $\Delta \vdash \lambda x.t : \sigma \to \tau$. □

For ϵ-reductions, we need some negative statements.

Lemma 15 None of the following judgments hold:

(1) $\Delta \nvdash c : \sigma \to \tau$
(2) $\Delta \nvdash (u_1, u_2) : \iota$, $\Delta \nvdash (u_1, u_2) : \sigma \to \tau$
(3) $\Delta \nvdash \mathsf{e} : \sigma \to \tau$, $\Delta \nvdash \mathsf{e} : \iota$, $\Delta \nvdash \mathsf{e} : \sigma\times\tau$
(4) $\Delta \nvdash c : \mathsf{E}$, $\Delta \nvdash f : \mathsf{E}$, $\Delta \nvdash \lambda x.t : \mathsf{E}$, $\Delta \nvdash (u_1, u_2) : \mathsf{E}$

Proof By contradiction. Let $\Delta \vdash v : \rho$ has the shortest proof for the pair v and ρ from some of the statements above. By their form, only ($\cup$L), ($\exists$L) and ($\trianglelefteq$) are applicable and ($\cup$L) and ($\exists$L) require a shorter proof. Lemma 12 applies for the form of v, and it is easy to show that ($\trianglelefteq$) also requires a shorter proof of the same form. □

Theorem 16 (Subject Reduction) If $\Delta \vdash t : \tau$ and $t \longrightarrow t'$, then $\Delta \vdash t' : \tau$.

Proof By induction on the proof of $\Delta \vdash t : \tau$.

Cases (ASS), (BASE), (PRIM) or (E). Not applicable because t is irreducible.

Cases ($\to$I), ($\times$I), ($\trianglelefteq$) or ($\forall$I). Obvious from the I.H.

Case ($\to$E) $\dfrac{\Delta \vdash r : \sigma \to \tau \qquad \Delta \vdash s : \sigma}{\Delta \vdash rs : \tau}$ with $t = rs$. We prove this case by induction on $t \longrightarrow t'$.

Case $r \longrightarrow r'$ and $t' = r's$. By the I.H. we have $\Delta \vdash r' : \sigma \to \tau$ and applying ($\to$E) we get $\Delta \vdash r's : \tau$.

Case $s \longrightarrow s'$ and $t' = rs'$. Analogous.

Case (β_{v}) $r = \lambda x.t''$, $s = v$ and $t' = t''[x \mapsto v]$. By Lemma 14, we have $\Delta,\ x : \sigma \vdash t'' : \tau$. Applying Lemma 13 we get $\Delta \vdash t''[x \mapsto v] : \tau$.

Case (δ) $r = f$, $s = v$ and $t' = \delta(f, v)$. This case is ensured by the δ-typability.

Case (ϵ) $t' = \mathsf{e}$ and r is in form c or (u, v) or e. It contradicts because $\nvdash r : \sigma \to \tau$ by Lemma 15.

Case ($\to^{\mathsf{C}}$E) $t = rs$, $\Delta \vdash r : (\top \to \top)^{\mathsf{C}}$, $\Delta \vdash s : \sigma$ and $\tau = \mathsf{E}$.

If $t' = r's$ or $t' = rs'$ with $r \longrightarrow r'$ or $s \longrightarrow s'$, it is obvious from the I.H.

Otherwise r must be a value. By the form and Lemma 12, we have to consider only ($\trianglelefteq$-$\to^{\mathsf{C}}$), i.e., $\vdash r : \top \to \top^{\mathsf{C}}$. Now the proof proceeds to a case analysis of $t \longrightarrow t'$.

Case (β_{v}) $r = \lambda x.t''$ and $t' = t''[x \mapsto s]$. By Lemma 14, we have $\Delta,\ x : \top \vdash t'' : \top^{\mathsf{C}}$. Using ($\trianglelefteq$L) with $\vdash \sigma \trianglelefteq \top$, we get $\Delta,\ x : \sigma \vdash t'' : \top^{\mathsf{C}}$, and using ($\subseteq$) with $\vdash \top^{\mathsf{C}} \subseteq \mathsf{E}$, we get $\Delta,\ x : \sigma \vdash t'' : \mathsf{E}$. By Lemma 13, $\Delta \vdash t' : \mathsf{E}$.

Case (δ). This case is ensured by δ-typability.

Case (ϵ). We have $t' = \mathsf{e}$ and $\Delta \vdash t' : \mathsf{E}$ by (E). □

5. Types of Expressions That Always/Never Cause Runtime Errors

As a corollary of the subject reduction theorem, the strong soundness is immediately obtained:

Theorem 17 (Strong Soundness) If $\vdash t : \tau$ and $t \longrightarrow\!\!\!\!\!\to v$, then $\vdash v : \tau$. □

The following *error soundness* ensures that expressions typed with E always cause runtime errors whenever they are evaluated.

Theorem 18 (Error Soundness) If $\vdash t : \mathsf{E}$ and $t \longrightarrow\!\!\!\!\!\to v$, then $v = \mathsf{e}$.

Proof By the strong soundness theorem, we have $\Delta \vdash v : \mathsf{E}$. Lemma 15–4 denies every possible form of v but e. □

Since our system types erroneous expressions, the usual reasoning of "typed expressions are safe" is of course unsound. To ensure the safety of typed expressions, we must restrict their types to *safe types*, which are not inhabited by e and pairs containing e.

Definition 19 (Safe Types) The set $\mathcal{T}_{safe}$ of *safe types* is the least set satisfying:

- $\iota, \sigma \to \tau \in \mathcal{T}_{safe}$ for any type σ and τ
- $\tau_1, \tau_2 \in \mathcal{T}_{safe} \implies \tau_1{\times}\tau_2, \tau_1 \cup \tau_2, \tau_1 \cap \tau_2 \in \mathcal{T}_{safe}$
- $\tau \in \mathcal{T}_{safe} \implies \mu\alpha.\tau, \forall\alpha.\tau \in \mathcal{T}_{safe}$

Note that we consider $\sigma \to \tau$ to be always safe, because any function does not raise a runtime error until it is applied. On the other hand, since a pair containing an error should be treated as an error, $\tau_1{\times}\tau_2$ is safe only if both τ_1 and τ_2 are safe.

Lemma 20 If τ is safe, then $\tau[\alpha \mapsto \sigma]$ is also safe.

Proof Obvious by structural induction on τ. □

Lemma 21 If $\vdash v : \tau$ for a safe type τ, then v is a safe value.

Proof By induction on the height of the proof of $\vdash v : \tau$. By the definition we have following cases of the form of τ:

- ι or $\rho \to \sigma$. By Lemma 15, v must be in form c or $\lambda x.t$, thus $v \in \mathcal{T}_{safe}$.
- $\sigma_1{\times}\sigma_2$ with σ_1 and σ_2 safe. In this case v must be in form (u_1, u_2) with $\vdash u_i : \sigma_i$ for both $i \in \{1, 2\}$. By the I.H. u_i is a safe value, thus so is (u_1, u_2).
- $\sigma_1 \cap \sigma_2$ or $\sigma_1 \cup \sigma_2$ with σ_1 and σ_2 safe. By Lemma 12 we get a shorter proof of $\vdash v : \tau'$ for either or both of $\tau' = \sigma_1$ and σ_2. Thus by the I.H., v is safe.
- $\mu\alpha.\sigma$ or $\forall\alpha.\sigma$ with σ safe. By Lemma 12 we have a shorter proof of $\vdash v : \tau'$ for $\tau' = \sigma[\alpha \mapsto \mu\alpha.\sigma]$ or $\sigma[\alpha \mapsto \rho]$. In either case, τ' is also safe by Lemma 20 and v is safe by the I.H. □

Theorem 22 (Weak Soundness) If $\vdash t : \tau$ and $t \longrightarrow\!\!\!\!\!\to v$ for a safe type τ, then v is a safe value.

$$\mathcal{D}: \left\{ \dfrac{\dfrac{}{\cdots \vdash f : \tau \to \tau}(\textsc{ass}) \quad \dfrac{\dfrac{}{\cdots \vdash x : \mu\alpha.\,\alpha \to \tau}(\textsc{ass}) \quad \dfrac{\dfrac{}{\cdots \vdash x : \mu\alpha.\,\alpha \to \tau}(\textsc{ass})}{\cdots \vdash x : (\mu\alpha.\,\alpha \to \tau) \to \tau}(\trianglelefteq\text{-}\mu\mathrm{E})}{\cdots \vdash xx : \tau}(\to\mathrm{E})}{\dfrac{\cdots, x : \mu\alpha.\,\alpha \to \tau \vdash f(xx) : \tau}{\Gamma,\ f : \tau \to \tau \vdash \lambda x.\,f(xx) : (\mu\alpha.\,\alpha \to \tau) \to \tau}(\to\mathrm{I})}(\to\mathrm{E}) \right.$$

in

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{\cdots \vdash \lambda x.\,f(xx) : (\mu\alpha.\,\alpha \to \tau) \to \tau} \quad \dfrac{\dfrac{\mathcal{D}}{\cdots \vdash \lambda x.\,f(xx) : (\mu\alpha.\,\alpha \to \tau) \to \tau}}{\cdots \vdash \lambda x.\,f(xx) : \mu\alpha.\,\alpha \to \tau}(\trianglelefteq\text{-}\mu\mathrm{I})}{\Gamma,\ f : \tau \to \tau \vdash (\lambda x.\,f(xx))(\lambda x.\,f(xx)) : \tau}(\to\mathrm{E})}{\Gamma \vdash \lambda f.(\lambda x.\,f(xx))(\lambda x.\,f(xx)) : (\tau \to \tau) \to \tau}(\to\mathrm{I})$$

Fig. 4 Admissibility of axiom (Y).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdots \vdash \mathtt{bool?} : \mathtt{Bool}^{\mathsf{C}} \to \mathtt{False}}(\textsc{prim}) \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\cdots \vdash x : \tau_1^{\mathsf{C}}}(\textsc{ass})}{\cdots \vdash x : (\mathtt{Bool} \cup (\mathtt{Bool} \to \tau_1))^{\mathsf{C}}}(\trianglelefteq)}{\cdots \vdash x : \mathtt{Bool}^{\mathsf{C}} \cap (\mathtt{Bool} \to \tau_1)^{\mathsf{C}}}(\trianglelefteq)}{\cdots \vdash x : \mathtt{Bool}^{\mathsf{C}}}(\trianglelefteq)}{\dfrac{\cdots \vdash \mathtt{bool?}\ x : \mathtt{False}}{\cdots \vdash \mathtt{if}\,(\mathtt{bool?}\ x) : \forall\alpha\beta.\,\alpha \to \beta \to \beta}(\to\mathrm{E})}(\to\mathrm{E})}{\cdots \vdash \mathtt{if}\,(\mathtt{bool?}\ x) : \tau_1^{\mathsf{C}} \to \mathsf{E} \to \mathsf{E}}(\subseteq) \quad \dfrac{}{\cdots \vdash x : \tau_1^{\mathsf{C}}}(\textsc{ass})}{\cdots \vdash \mathtt{if}\,(\mathtt{bool?}\ x)\ x : \mathsf{E} \to \mathsf{E}}(\to\mathrm{E}) \quad \dfrac{\dfrac{}{\cdots \vdash \mathtt{or} : (\mathtt{Bool}{\times}\mathtt{Bool})^{\mathsf{C}} \to \mathsf{E}}(\textsc{prim}) \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\cdots \vdash f : \tau_1^{\mathsf{C}} \to \mathsf{E}}(\textsc{ass}) \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdots \vdash x : \tau_1^{\mathsf{C}}}(\textsc{ass})}{\cdots \vdash x : (\mathtt{Bool} \cup (\mathtt{Bool} \to \tau_1))^{\mathsf{C}}}(\trianglelefteq)}{\cdots \vdash x : \mathtt{Bool}^{\mathsf{C}} \cap (\mathtt{Bool} \to \tau_1)^{\mathsf{C}}}(\trianglelefteq)}{\cdots \vdash x : \mathtt{Bool}^{\mathsf{C}} \cap (\mathtt{Bool} \to \tau_1^{\mathsf{C}})}(\trianglelefteq)}{\cdots \vdash x : \mathtt{Bool} \to \tau_1^{\mathsf{C}}}(\trianglelefteq) \quad \dfrac{}{\cdots \vdash \mathtt{true} : \mathtt{Bool}}(\textsc{base})}{\cdots \vdash (x\ \mathtt{true}) : \tau_1^{\mathsf{C}}}(\to\mathrm{E})}{\cdots \vdash f(x\ \mathtt{true}) : \mathsf{E}}(\to\mathrm{E}) \quad \dfrac{\vdots}{\cdots \vdash f(x\ \mathtt{false}) : \mathsf{E}}}{\cdots \vdash (f(x\ \mathtt{true}), f(x\ \mathtt{false})) : \mathsf{E}{\times}\mathsf{E}}(\times\mathrm{I})}{\cdots \vdash (f(x\ \mathtt{true}), f(x\ \mathtt{false})) : (\mathtt{Bool}{\times}\mathtt{Bool})^{\mathsf{C}}}(\subseteq)}{\cdots \vdash \mathtt{or}(f(x\ \mathtt{true}), f(x\ \mathtt{false})) : \mathsf{E}}(\to\mathrm{E})}{f : \tau_1^{\mathsf{C}} \to \mathsf{E},\ x : \tau_1^{\mathsf{C}} \vdash t_0 : \mathsf{E}}(\to\mathrm{E})$$

$$\dfrac{\dfrac{}{\vdash \mathsf{Y} : ((\tau_1^{\mathsf{C}} \to \mathsf{E}) \to \tau_1^{\mathsf{C}} \to \mathsf{E}) \to \tau_1^{\mathsf{C}} \to \mathsf{E}}(\mathsf{Y}) \quad \dfrac{\dfrac{f : \tau_1^{\mathsf{C}} \to \mathsf{E},\ x : \tau_1^{\mathsf{C}} \vdash t_0 : \mathsf{E}}{f : \tau_1^{\mathsf{C}} \to \mathsf{E} \vdash \lambda x.\,t_0 : \tau_1^{\mathsf{C}} \to \mathsf{E}}(\to\mathrm{I})}{\vdash \lambda f x.\,t_0 : (\tau_1^{\mathsf{C}} \to \mathsf{E}) \to \tau_1^{\mathsf{C}} \to \mathsf{E}}(\to\mathrm{I})}{\vdash \mathsf{Y}\ \lambda f x.\,t_0 : \tau_1^{\mathsf{C}} \to \mathsf{E}}(\to\mathrm{E})$$

Fig. 5 Derivation of $\mathtt{SAT} : \tau_1^{\mathsf{C}} \to \mathsf{E}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash (\mathtt{Bool}{\times}\mathtt{Bool})^{\mathsf{C}} \to \mathsf{E} \trianglelefteq \mathtt{Bool}^{\mathsf{C}}}(\trianglelefteq\text{-}{\to}\mathcal{B}^{\mathsf{C}}) \quad \dfrac{\dfrac{}{\vdash \mathtt{Bool} \trianglelefteq (\mathtt{Bool}{\times}\mathtt{Bool})^{\mathsf{C}}}(\trianglelefteq\text{-}\mathcal{B}{\times}^{\mathsf{C}}) \quad \dfrac{\vdots}{\vdash \mathsf{E} \subseteq \tau_1^{\mathsf{C}}}}{\vdash (\mathtt{Bool}{\times}\mathtt{Bool})^{\mathsf{C}} \to \mathsf{E} \subseteq \mathtt{Bool} \to \tau_1^{\mathsf{C}}}(\trianglelefteq\text{-}{\to})}{\vdash ((\mathtt{Bool}{\times}\mathtt{Bool})^{\mathsf{C}} \to \mathsf{E}) \cap ((\mathtt{Bool}{\times}\mathtt{Bool})^{\mathsf{C}} \to \mathsf{E}) \subseteq \mathtt{Bool}^{\mathsf{C}} \cap \mathtt{Bool} \to \tau_1^{\mathsf{C}}}}{\vdash (\mathtt{Bool}{\times}\mathtt{Bool})^{\mathsf{C}} \to \mathsf{E} \subseteq \mathtt{Bool}^{\mathsf{C}} \cap \mathtt{Bool} \to \tau_1^{\mathsf{C}}}}{\vdash (\mathtt{Bool}{\times}\mathtt{Bool})^{\mathsf{C}} \to \mathsf{E} \subseteq \tau_1^{\mathsf{C}}}$$

Fig. 6 Derivation of $\vdash \tau_2 \subseteq \tau_1^{\mathsf{C}}$.

Proof By the strong soundness, we have $\vdash v : \tau$. Since τ is assumed safe, Lemma 21 ensures that v is a safe value. □

6. Examples

In this section we demonstrate how our type system detects runtime errors by taking the Scheme function SAT in the introduction, for example. To express SAT in our language, we need a fixpoint operator $\mathsf{Y} := \lambda f.(\lambda x.\,f(xx))(\lambda x.\,f(xx))$.

Proposition 23 The following axiom is admissible:

$$\Delta \vdash \mathsf{Y} : (\tau \to \tau) \to \tau \qquad (\mathsf{Y})$$

Proof This is originally shown by MacQueen et al. [17] The proof tree for our system is shown in **Fig. 4**. □

Using Y, SAT is represented in Λ by $\mathsf{Y}\ \lambda f x.\,t_0$ where

$$t_0 = \mathtt{if}(\mathtt{bool?}\ x)\ x\ (\mathtt{or}(f(x\ \mathtt{true}), f(x\ \mathtt{false})))$$

For function symbols appearing here, we only consider their types:

$$\mathsf{Ty}(\mathtt{if}) = \begin{cases} \mathtt{True} \to (\forall\alpha\beta.\,\alpha \to \beta \to \alpha) \\ \mathtt{False} \to (\forall\alpha\beta.\,\alpha \to \beta \to \beta) \end{cases}$$

$$\mathsf{Ty}(\mathtt{or}) = \begin{cases} \mathtt{Bool}{\times}\mathtt{Bool} \to \mathtt{Bool} \\ (\mathtt{Bool}{\times}\mathtt{Bool})^{\mathsf{C}} \to \mathsf{E} \end{cases}$$

$$\mathsf{Ty}(\mathtt{bool?}) = \begin{cases} \mathtt{Bool} \to \mathtt{True} \\ \mathtt{Bool}^{\mathsf{C}} \to \mathtt{False} \end{cases}$$

where True, False and Bool are base types with $C_{\mathtt{True}} = \{\mathtt{true}\}$, $C_{\mathtt{False}} = \{\mathtt{false}\}$ and $C_{\mathtt{Bool}} = C_{\mathtt{True}} \cup C_{\mathtt{False}}$. Now we can deduce $\vdash \mathtt{SAT} : (\tau_1 \to \mathtt{Bool}) \cap (\tau_1^{\mathsf{C}} \to \mathsf{E})$ for $\tau_1 = \mu\alpha.\,\mathtt{Bool} \cup (\mathtt{Bool} \to \alpha)$, as the following proof tree shows:

$$\dfrac{\dfrac{\vdots}{\vdash \mathtt{SAT} : \tau_1 \to \mathtt{Bool}} \quad \dfrac{\text{Fig. 5}}{\vdash \mathtt{SAT} : \tau_1^{\mathsf{C}} \to \mathsf{E}}}{\vdash \mathtt{SAT} : (\tau_1 \to \mathtt{Bool}) \cap (\tau_1^{\mathsf{C}} \to \mathsf{E})}(\cap\mathrm{I})$$

The subproof for $\vdash \mathtt{SAT} : {\tau_1}^{\mathsf{C}} \to \mathsf{E}$ is presented in **Fig. 5**. As described in the introduction, this typing means that SAT returns a boolean value for any curried boolean function and boolean input, and always causes a runtime error for other inputs.

An uncurried binary boolean function t_2 should have following type τ_2:

$$\tau_2 = (\mathtt{Bool}{\times}\mathtt{Bool} \to \mathtt{Bool}) \cap ((\mathtt{Bool}{\times}\mathtt{Bool})^{\mathsf{C}} \to \mathsf{E})$$

We have $\vdash \tau_2 \subseteq \tau_1^{\mathsf{C}}$ as shown in **Fig. 6**, thus we can deduce $\vdash \mathtt{SAT}\ t_2 : \mathsf{E}$. This typing means that the evaluation of $\mathtt{SAT}\ t_2$ causes a runtime error.

7. Related Works

Intersection types were introduced by Coppo et al. [7] and Pottinger [20]. They showed that reduction preserves typing (often called *Subject Reduction* after Curry), and that an expression is

Table 1 Expressiveness of types.

	$\sigma \cap \tau$	$\sigma \cup \tau$	$\forall\alpha.\tau$	$\exists\alpha.\tau$	$\mu\alpha.\tau$	$\sigma \subseteq \tau$	τ^{C}	E	Type Inference Algorithm
Hindley/Milner [14], [18]			√						√
Intersection Types [7], [20]	√								
Barendregt et al. [4]	√					√			
MacQueen et al. [17]	√	√	√	√	√				
Barbanera et al. [3]	√	√				√			
Soft Typing [5]		√	√		√	√			√
Amadio et al. [2]					√	√			√
Damm [8]	√	√			√	√			
Aiken et al. [1]	√	√				√			√*2
Semantic Subtyping [11]	√	√			√	√	√		√*3
Hosoya et al. [15]	√	√	√		√	√	√		√*3
Our system	√	√	√	√	√	√	√	√	

typable if and only if it is strongly normalizing. This means that a complete type inference algorithm does not exist for intersection types.

Milner [18] presented a soundness theorem for Hindley/Milner style polymorphism by means of denotational semantics. Types are interpreted as downward-closed and directed-complete sets (*ideals*) in the semantic domain of expressions. He also presented the famous type inference algorithm $\mathcal{W}$ which is complete w.r.t. first rank polymorphism.

MacQueen et al. [17] extended the ideal model for *recursive types*, and additionally introduced unquoted *existential quantification* and set-theoretic *union types*. They formalized the semantics of a recursive type as the unique fixpoint of corresponding contractive mapping on ideals. They restricted the types to be *formally contractive* in order to ensure the fixpoint is unique. (This harmless restriction was eliminated anyway in our system.)

Subtyping on intersection types was presented by Barendregt et al. [4], and later extended for intersection and union types by Barbanera et al. [3].

Amadio et al. [2] formalized subtyping on recursive types with the help of regular tree expressions, and the system was extended for union, intersection and recursive types by Damm [8]. Subtyping on union and recursive types is also presented by Cartwright et al. [5].

Frisch et al. [11], [12] introduced *semantic subtyping*. They presented a set-theoretic model of types independent from the semantics of expressions, and defined the subtyping relation by the set inclusion relation on this model. The semantic subtyping method allowed set-theoretic interpretation of *complement type* τ^{C}, which was not possible in the ideal model [6]. Hosoya et al. [15] extended the semantic subtyping for parametric polymorphism.

Our system covers all the notions described above, except that we have not yet proposed a type inference algorithm. These results are summarized in **Table 1**.

Several studies have been motivated to cover the disadvantages of dynamically typed languages by means of static typing.

Soft typing [1], [5], [25] *Soft typing* introduces a static type check for dynamically typed languages. If the static type check succeeds, the program is assured to be type-safe. In order not to restrict the expressiveness of dynamically typed languages, programs will not be rejected even though the static check fails; instead, runtime checks are inserted which are unknown to fail or not on execution. That is, soft typing does not detect runtime errors; programmers must manually check whether a 'softly' rejected program contains a bug, or not.

Complete typing [23] *Complete typing* rejects provably erroneous programs at compile time, and is expected to help detect real bugs from softly rejected programs. Complete typing requires an inference system that is different from the usual sound system, and it requires an algorithm testing *disjointness* of types (i.e. *emptiness of intersection types*), which is known to be undecidable [21].

Hybrid typing [10], [16] *Hybrid typing* is an extension of soft typing with *refinement types*, and it rejects some programs at compile time if the type check is proved to fail. However, hybrid type checking is not suitable for our purpose because it was originally designed for a statically typed language, and type check failure does not immediately imply a runtime error.

8. Conclusions and Future Work

We have presented a type system that accepts expressions whose evaluation *never* causes runtime errors, and rejects expressions whose evaluation *always* causes runtime errors. The system is proved to be sound by several soundness theorems:

Subject Reduction If an expression has a type, then its reduct also has the same type. (Theorem 16)

Strong Soundness If an expression has a type and its evaluation terminates, then the value has the same type. (Theorem 17)

Weak Soundness If an expression has a safe type, then its evaluation never causes a runtime error. That is, the expression should be accepted by the type checker. (Theorem 22)

Error Soundness If an expression has the type E, then its evaluation must cause a runtime error. That is, the expression should be rejected by the type checker. (Theorem 18)

Though we have presented the type system, we have not yet presented a type inference algorithm. Since our system has in-

*2 Type inference is undecidable in their system. An incomplete algorithm was presented [1].

*3 These algorithms are complete w.r.t. semantic subtyping, but not for the axiomatized intersection types [11], [15].

tersection types and higher rank polymorphism where a complete type inference is impossible for each system [19], [22], a complete type inference is apparently impossible. Thus, we need to discover a fragment of our system where type inference becomes possible in practical time.

Reference

[1] Aiken, A., Wimmers, E.L. and Lakshman, T.K.: Soft typing with conditional types, *Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, *POPL '94*, pp.163–173 (1994).

[2] Amadio, R. and Cardelli, L.: Subtyping Recursive Types, *ACM Trans. Prog. Lang. Syst.*, Vol.15, No.4, pp.575–631 (1993).

[3] Barbanera, F. and Dezani-Ciancaglini, M.: Intersection and union types, Theoretical Aspects of Computer Software, Ito, T. and Meyer, A. (Eds.), *Lecture Notes in Computer Science*, Vol.526, pp.651–674, Springer Berlin/Heidelberg (1991).

[4] Barendregt, H., Coppo, M. and Dezani-Ciancaglini, M.: A Filter Lambda Model and the Completeness of Type Assignment, *J. Symbolic Logic*, Vol.48, No.4, pp.931–940 (1983).

[5] Cartwright, R. and Fagan, M.: Soft typing, *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp.278–292 (1991).

[6] Castagna, G. and Frisch, A.: A gentle introduction to semantic subtyping, *Proc. 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, *PPDP '05*, pp.198–199 (2005).

[7] Coppo, M. and Dezani-Ciancaglini, M.: An Extension of the Basic Functionality Theory for the λ-calculus, *Nortre Dame Journal of Formal Logic*, Vol.21, No.4, pp.685–693 (1980).

[8] Damm, F.: Subtyping with union types, intersection types and recursive types, Theoretical Aspects of Computer Software, Hagiya, M. and Mitchell, J. (Eds.), *Lecture Notes in Computer Science*, Vol.789, pp.687–706, Springer Berlin/Heidelberg (1994).

[9] Dezani-Ciancaglini, M., de'Liguoro, U. and Piperno, A.: Filter models for conjunctive-disjunctive λ-calculi, *Theor. Comput. Sci.*, Vol.170, No.1–2, pp.83–128 (1996).

[10] Flanagan, C.: Hybrid type checking, *SIGPLAN Notices*, Vol.41, pp.245–256 (2006).

[11] Frisch, A., Castagna, G. and Benzaken, V.: Semantic subtyping, *17th Annual IEEE Symposium on Logic in Computer Science*, pp.137–146 (2002).

[12] Frisch, A., Castagna, G. and Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types, *J. ACM*, Vol.55, pp.19:1–19:64 (2008).

[13] Girard, J.-Y.: The system F of variable types, fifteen years later, *Theor. Comput. Sci.*, Vol.45, pp.159–192 (1986).

[14] Hindley, R.: The Principal Type-Scheme of an Object in Combinatory Logic, *Transactions of the American Mathematical Society*, Vol.146, pp.29–60 (1969).

[15] Hosoya, H., Frisch, A. and Castagna, G.: Parametric polymorphism for XML, *SIGPLAN Not.*, Vol.40, pp.50–62 (2005).

[16] Knowles, K. and Flanagan, C.: Hybrid type checking, *ACM Trans. Program. Lang. Syst.*, Vol.32, pp.6:1–6:34 (2010).

[17] MacQueen, D., Plotkin, G. and Sethi, R.: An ideal model for recursive polymorphic types, *Proc. 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, *POPL '84*, New York, NY, USA, pp.165–174, ACM (1984).

[18] Milner, R.: A Theory of Type Polymorphism in Programming, *J. Comput. Syst. Sci.*, pp.348–375 (1978).

[19] Pierce, B.C.: *Types and Programming Languages*, The MIT Press (2002).

[20] Pottinger, G.: A type assignment for the strongly normalizable λ-terms, *To H.B. Curry : Essays on combinatory logic, lambda calculus, and formalism*, Seldin, J. and Hindley, J. (Eds.), pp.561–577, Academic Press (1980).

[21] Urzyczyn, P.: The Emptiness Problem for Intersection Types, *J. Symbolic Logic*, Vol.64, No.3, pp.1195–1215 (1999).

[22] Wells, J.B.: Typability and type checking in System F are equivalent and undecidable, *Annals of Pure and Applied Logic*, Vol.98, No.1–3, pp.111–156 (1999).

[23] Widera, M.: A Sketch of Complete Type Inference for Functional Programming, *International Workshop on Functional and* (*Constraint*) *Logic Programming* (*WLFP 2001*) (2001).

[24] Widera, M. and Beierle, C.: An Approach to Checking the Non-Disjointness of Types in Functional Programming, *Infomatik Berichte 281*, FernUniversität Hagen (2001).

[25] Wright, A.K. and Cartwright, R.: A Practical Soft Type System for Scheme, *ACM Trans. Prog. Lang. Syst.*, Vol.19, No.1, pp.87–152 (1997).

[26] Wright, A.K. and Felleisen, M.: A Syntactic Approach to Type Soundness, *Information and Computation*, Vol.115, pp.38–94 (1992).

Appendix

A.1 Detailed Proofs

Lemma 11 Let $\alpha, \alpha' \notin \mathsf{FV}(\Gamma)$, $\alpha \notin \mathsf{FV}(\tau')$ and $\alpha' \notin \mathsf{FV}(\tau)$. If $\Gamma \vdash \sigma \trianglelefteq \sigma'$ and $\Gamma,\ \alpha \trianglelefteq \alpha' \vdash \tau \trianglelefteq \tau'$ then $\Gamma \vdash \tau[\alpha \mapsto \sigma] \trianglelefteq \tau'[\alpha' \mapsto \sigma']$.

Proof By induction on the height of the proof of $\Gamma,\ \alpha \trianglelefteq \alpha' \vdash \tau \trianglelefteq \tau'$.

Case ($\trianglelefteq$-ASS). The following two subcases are possible:

Case $\tau = \alpha$ and $\tau' = \alpha'$. Then the assumption $\Gamma \vdash \sigma \trianglelefteq \sigma'$ is equivalent to $\Gamma \vdash \tau[\alpha \mapsto \sigma] \trianglelefteq \tau'[\alpha' \mapsto \sigma']$.

Case $(\beta \trianglelefteq \beta') \in \Gamma$ with $\tau = \beta$ and $\tau' = \beta'$. Since $\alpha, \alpha' \notin \mathsf{FV}(\Gamma)$, by ($\trianglelefteq$-ASS) we get $\Gamma \vdash \beta[\alpha \mapsto \sigma] \trianglelefteq \beta'[\alpha' \mapsto \sigma']$.

Case ($\trianglelefteq$-μC) $\dfrac{\Gamma,\ \alpha \trianglelefteq \alpha',\ \beta \trianglelefteq \beta' \vdash \rho \trianglelefteq \rho'}{\Gamma,\ \alpha \trianglelefteq \alpha' \vdash \mu\beta.\rho \trianglelefteq \mu\beta'.\rho'}$ with $\tau = \mu\beta.\rho$ and $\tau' = \mu\beta'.\rho'$. Since $\alpha \notin \mathsf{FV}(\rho')$ and $\alpha' \notin \mathsf{FV}(\rho)$, by the I.H. we have $\Gamma,\ \beta \trianglelefteq \beta' \vdash \rho[\alpha \mapsto \sigma] \trianglelefteq \rho'[\alpha' \mapsto \sigma']$. Applying ($\trianglelefteq$-$\mu$C), we get $\Gamma \vdash \mu\beta.\rho[\alpha \mapsto \sigma] \trianglelefteq \mu\beta'.\rho'[\alpha' \mapsto \sigma']$.

Others. Obvious because they are independent from the subtyping environment and closed under type substitution. □

Lemma 12. If $\Delta \vdash v : \tau$ has a proof with height h, then there exists a proof shorter than h for:

(1) both $\Delta \vdash v : \tau_1$ and $\Delta \vdash v : \tau_2$ if $\tau = \tau_1 \cap \tau_2$,

(2) either $\Delta \vdash v : \tau_1$ or $\Delta \vdash v : \tau_2$ if $\tau = \tau_1 \cup \tau_2$,

(3) $\Delta \vdash v : \sigma[\alpha \mapsto \mu\alpha.\sigma]$ if $\tau = \mu\alpha.\sigma$,

(4) $\Delta \vdash v : \sigma[\alpha \mapsto \rho]$ if $\tau = \forall\alpha.\sigma$.

Proof By mutual induction on the proof of $\Delta \vdash v : \tau$.

Case ($\cap$I) $\dfrac{\Delta \vdash v : \tau_1 \qquad \Delta \vdash v : \tau_2}{\Delta \vdash v : \tau_1 \cap \tau_2}$. This case applies for statement 1 and it is trivial.

Cases ($\cup$L) or ($\exists$L). All the statements are immediate from the I.H.

Case ($\forall$I) $\dfrac{\Delta \vdash v : \sigma}{\Delta \vdash v : \forall\alpha.\sigma}$ with $\alpha \notin \mathsf{FV}(\Delta)$. This case applies for statement 4. By replacing α to ρ in the proof of $\Delta \vdash v : \sigma$, we can make a proof of height $h-1$ for $\Delta \vdash v : \sigma[\alpha \mapsto \rho]$.

Case ($\trianglelefteq$) $\dfrac{\Delta \vdash v : \tau'}{\Delta \vdash v : \tau}$ with $\vdash \tau' \trianglelefteq \tau$. Proof proceeds to case analysis of $\vdash \tau' \trianglelefteq \tau$.

- The following subcases apply for all statements:

Case ($\trianglelefteq$-$\cup$E) $\dfrac{\Delta \vdash v : \tau \cup \tau}{\Delta \vdash v : \tau}$. By I.H. 2, $\Delta \vdash v : \tau$ has a proof shorter than $h-1$.

Case ($\trianglelefteq$-$\cap$E) $\dfrac{\Delta \vdash v : \sigma \cap \tau}{\Delta \vdash v : \tau}$. By I.H. 1, $\Delta \vdash v : \tau$ has a proof shorter than $h-1$.

Case ($\trianglelefteq$-μE) $\dfrac{\Delta \vdash v : \mu\alpha.\sigma}{\Delta \vdash v : \tau}$ with $\tau = \sigma[\alpha \mapsto \mu\alpha.\sigma]$. By I.H. 3, $\Delta \vdash v : \tau$ has a proof shorter than $h-1$.

Case ($\trianglelefteq$-$\forall$E) $\dfrac{\Delta \vdash v : \forall\alpha.\sigma}{\Delta \vdash v : \sigma[\alpha \mapsto \rho]}$ with $\tau = \sigma[\alpha \mapsto \rho]$. By I.H. 4, $\Delta \vdash v : \tau$ has a proof shorter than $h-1$.

In either case above we could reduce the statements to where the I.H.s can be applied.

- The following subcases apply for statement 1;

  Case ($\trianglelefteq$-$\cap$I) $\dfrac{\Delta \vdash v : \tau_1}{\Delta \vdash v : \tau_1 \cap \tau_1}$ with $\tau_1 = \tau_2$. Trivial.

  Case ($\trianglelefteq$-$\cap$C) $\dfrac{\Delta \vdash v : \tau_1' \cap \tau_2'}{\Delta \vdash v : \tau_1 \cap \tau_2}$ with $\vdash \tau_1' \trianglelefteq \tau_1$ and $\vdash \tau_2' \trianglelefteq \tau_2$. By I.H. 1, $\Delta \vdash v : \tau_i'$ with a proof shorter than $h - 1$, for all $i \in \{1, 2\}$. Applying ($\trianglelefteq$) we get a proof of $\Delta \vdash v : \tau_i$, which is shorter than h.

- The following subcases apply for statement 2;

  Case ($\trianglelefteq$-$\cup$I) $\dfrac{\Delta \vdash v : \tau_1}{\Delta \vdash v : \tau_1 \cup \tau_2}$. Trivial.

  Case ($\trianglelefteq$-$\cup$C) $\dfrac{\Delta \vdash v : \tau_1' \cup \tau_2'}{\Delta \vdash v : \tau_1 \cup \tau_2}$ with $\vdash \tau_1' \trianglelefteq \tau_1$ and $\vdash \tau_2' \trianglelefteq \tau_2$. By I.H. 2, $\Delta \vdash v : \tau_i'$ has a proof shorter than $h - 1$ for some $i \in \{1, 2\}$. So applying ($\trianglelefteq$) gives $\Delta \vdash v : \tau_i$ a proof shorter than h.

  Case ($\trianglelefteq$-$\cap\cup$) $\dfrac{\Delta \vdash v : \rho \cap (\sigma_1 \cup \sigma_2)}{\Delta \vdash v : (\rho \cap \sigma_1) \cup (\rho \cap \sigma_2)}$ with $\tau_1 = \rho \cap \sigma_1$ and $\tau_2 = \rho \cap \sigma_2$.

  By I.H. 1, we have a proof shorter than $h - 1$ for both

  $$\Delta \vdash v : \rho \qquad \text{and} \tag{A.1}$$
  $$\Delta \vdash v : \sigma_1 \cup \sigma_2 \tag{A.2}$$

  By I.H. 2 on (A.2), we have a proof shorter than $h - 2$ for

  $$\Delta \vdash v : \sigma_i \quad \text{for some } i \in \{1, 2\} \tag{A.3}$$

  Applying ($\cap$I) with (A.1) and (A.3), we get a proof shorter than h of $\Delta \vdash v : \tau_i$.

- The following subcases apply for statement 3;

  Case ($\trianglelefteq$-μI) $\dfrac{\Delta \vdash v : \sigma[\alpha \mapsto \mu\alpha.\,\sigma]}{\Delta \vdash v : \mu\alpha.\,\sigma}$. Trivial.

  Case ($\trianglelefteq$-μC) $\dfrac{\Delta \vdash v : \mu\alpha'.\,\sigma'}{\Delta \vdash v : \mu\alpha.\,\sigma}$ with $\alpha \notin \mathsf{FV}(\sigma')$, $\alpha' \notin \mathsf{FV}(\sigma)$ and $\alpha' \trianglelefteq \alpha \vdash \sigma' \trianglelefteq \sigma$. By Lemma 11 we have

  $$\Gamma \vdash \sigma'[\alpha' \mapsto \mu\alpha'.\,\sigma'] \trianglelefteq \sigma[\alpha \mapsto \mu\alpha.\,\sigma]. \tag{A.4}$$

  On the other hand, by I.H. 3 $\Delta \vdash v : \sigma'[\alpha' \mapsto \mu\alpha'.\,\sigma']$ has a proof shorter than $h - 1$.

  Applying ($\trianglelefteq$) with (A.4), we get a proof of $\Delta \vdash v : \sigma[\alpha \mapsto \mu\alpha.\,\sigma]$, which is shorter than h.

- No other subcases of subtyping apply because of the form τ.

Others. Not applicable because of the form of v or τ. □

Lemma 13 (Substitution) Let $x \notin \mathsf{Dom}(\Delta)$. If $\Delta,\ x : \sigma \vdash t : \tau$ and $\Delta \vdash v : \sigma$ then $\Delta \vdash t[x \mapsto v] : \tau$.

Proof By induction on the proof of $\Delta,\ x : \sigma \vdash t : \tau$.

Case (ASS). If $t = x$, then $\tau = \sigma$ and $t[x \mapsto v] = v$, so $\Delta \vdash v : \sigma$ is the goal. Otherwise $(t : \tau) \in \Delta$ and $t[x \mapsto v] = t$, so by (ASS) we get $\Delta \vdash t[x \mapsto v] : \tau$.

Cases (BASE), (PRIM), (E). Trivial.

Case ($\to$I) $\dfrac{\Delta,\ x : \sigma,\ x' : \sigma' \vdash t' : \tau'}{\Delta,\ x : \sigma \vdash \lambda x'.\,t' : \sigma' \to \tau'}$ with $t = \lambda x'.\,t'$ and $\tau = \sigma' \to \tau'$. Here we assumed that x' is properly renamed to avoid capture. By the I.H. we have $\Delta,\ x' : \sigma' \vdash t'[x \mapsto v] : \tau'$, and applying ($\to$I), we get $\Delta \vdash \lambda x'.\,t'[x \mapsto v] : \sigma' \to \tau'$.

Case ($\cup$L). There are two subcases to examine.

Case $\dfrac{\Delta' : x' : \sigma_1 : x : \sigma \vdash t : \tau : \Delta' : x' : \sigma_2 : x : \sigma \vdash t : \tau}{\Delta' : x' : \sigma_1 \cup \sigma_2 : x : \sigma \vdash t : \tau}$ with $\Delta = (\Delta',\ x' : \sigma_1 \cup \sigma_2)$. This case is obvious from the I.H.

Case $\dfrac{\Delta,\ x : \sigma_1 \vdash t : \tau \qquad \Delta,\ x : \sigma_2 \vdash t : \tau}{\Delta,\ x : \sigma_1 \cup \sigma_2 \vdash t : \tau}$ with $\sigma = \sigma_1 \cup \sigma_2$. By Lemma 12–2 on $\Delta \vdash v : \sigma_1 \cup \sigma_2$, we have either $\Delta \vdash v : \sigma_1$ or $\Delta \vdash v : \sigma_2$. In either case, by the I.H., we get $\Delta \vdash t[x \mapsto v] : \tau$.

Others. Obvious from the I.H. □

Lemma 14 (Abstraction) If $\Delta \vdash \lambda x.\,t : \sigma \to \tau$ then $\Delta,\ x : \sigma \vdash t : \tau$.

Proof By induction on $\Delta \vdash \lambda x.\,t : \sigma \to \tau$.

Case ($\to$I). Trivial.

Cases ($\cup$L) or ($\exists$L). Obvious from the I.H.

Case ($\trianglelefteq$). Because of the form $\sigma \to \tau$, the following subcases are possible;

Case ($\trianglelefteq$-$\to$) $\dfrac{\Delta \vdash \lambda x.\,t : \sigma' \to \tau'}{\Delta \vdash \lambda x.\,t : \sigma \to \tau}$ where $\vdash \sigma \trianglelefteq \sigma'$ and $\vdash \tau' \trianglelefteq \tau$. By the I.H. we have $\Delta,\ x : \sigma' \vdash t : \tau'$. Applying ($\trianglelefteq$) and ($\trianglelefteq$L) we get $\Delta,\ x : \sigma \vdash t : \tau$.

Cases ($\trianglelefteq$-$\cup$E), ($\trianglelefteq$-$\cap$E) ($\trianglelefteq$-μE) or ($\trianglelefteq$-$\forall$E). In either case, Lemma 12 gives a shorter proof for $\Delta \vdash \lambda x.\,t : \sigma \to \tau$. So by the I.H. we get $\Delta,\ x : \sigma \vdash t : \tau$.

Case ($\trianglelefteq$-$\to\cap$) $\dfrac{\Delta \vdash \lambda x.\,t : (\sigma \to \tau_1) \cap (\sigma \to \tau_2)}{\Delta \vdash \lambda x.\,t : \sigma \to \tau_1 \cap \tau_2}$. Lemma 12 gives a shorter proof of $\Delta \vdash \lambda x.\,t : \sigma \to \tau_i$ for both $i \in \{1, 2\}$. So by the I.H. we have $\Delta,\ x : \sigma \vdash t : \tau_i$. Applying ($\cap$I) we get $\Delta,\ x : \sigma \vdash t : \tau_1 \cap \tau_2$.

Case ($\trianglelefteq$-$\cup\to$) $\dfrac{\Delta \vdash \lambda x.\,t : (\sigma_1 \to \tau) \cap (\sigma_2 \to \tau)}{\Delta \vdash \lambda x.\,t : \sigma_1 \cup \sigma_2 \to \tau}$. Analogously we have $\Delta,\ x : \sigma_i \vdash t : \tau$. Applying ($\cup$L) we get $\Delta,\ x : \sigma_1 \cup \sigma_2 \vdash t : \tau$.

Others. Not applicable because of the form $\lambda x.\,t$ or $\sigma \to \tau$. □

Akihisa Yamada received his B.E. and M.E. from Nagoya University in 2006 and 2008, respectively. From 2008 to 2011 he worked for Panasonic Advanced Technology Development Co., Ltd. Since 2011, he has been a graduate student in the doctoral program of Graduate School of Information Science at Nagoya University. His research interests include type systems and term rewriting systems. He is a student member of IPSJ.

Keiichirou Kusakari received his B.E. from Tokyo Institute of Technology in 1994, received M.E. and Ph.D. degrees from Japan Advanced Institute of Science and Technology in 1996 and 2000. From 2000, he was a research associate at Tohoku University. He transferred to Nagoya University's Graduate School of Information Science in 2003 as an assistant professor and became an associate professor in 2006. His research interests include term rewriting systems, program theory, and automated theorem proving. He is a member of IEICE, IPSJ and JSSST.

Toshiki Sakabe was born in 1949. He received his B.E., M.E. and D.E. degrees from Nagoya University in 1972, 1974 and 1978, respectively. He was a research associate at Nagoya University during 1977–1985, and an associate professor at Mie University and Nagoya University during 1985–1987 and 1987–1993, respectively. He has been a professor of Nagoya University since 1993. His research interests are in the field of theoretical software science including algebraic specifications, rewriting computation, program verification, model checking and so on. He is a member of IPSJ, IEICE, JSAI and JSSST.

Masahiko Sakai completed graduate course of Nagoya University in 1989 and became an assistant professor, where he obtained a D.E. degree in 1992. From April 1993 to March 1997, he was an associate professor in JAIST, Hokuriku. In 1996 he stayed at SUNY at Stony Brook for six months as a visiting research professor. From April 1997, he was an associate professor in Nagoya University. Since December 2002, he has been a professor. He is interested in term rewriting system, verification of specification and software generation. He received the Best Paper Award from IEICE in 1992 and 2011. He is member of IEICE and JSSST.

Naoki Nishida graduated with a D.E. degree from a Graduate School of Engineering at Nagoya University in 2004. He became a research associate in Graduate School of Information Science at Nagoya University in 2004. From April 2007, he has been an Assistant Professor, and he was a visiting researcher in DSIC at Technical University of Valencia from July 2011 to December 2011. He received the Best Paper Award from IEICE in 2011. He is interested in program inversion, theorem proving, term rewriting, and program verification. He is a member of IEICE, JSSST and IPSJ.