

Full integration of subtyping and if-expression

Gang Chen^{*}
Computer Science Department
Boston University
Boston MA 02215 USA
Email: gangchen@types.bu.edu

ABSTRACT

The combination of full subsumption and conditional expression is a challenging problem, because in such a system, a term might not have a type which is a representative of all types of the term. Therefore, the traditional type checking technique fails. Due to such a difficulty, Java typing rule for if-expression uses only a restricted form of subtyping. In this paper, we propose a type system which includes both of the above mentioned features and enjoys decidable type checking. We also show that the system has the subject reduction property. It is expected that this technique could be used to improve the type system of Java with full subsumption.

Categories and Subject Descriptors

D.3 [Software]: Programming Languages

General Terms

Languages

Keywords

subtyping, if-expression, type checking, Java

1. INTRODUCTION

Subtyping is one of the essential features in object oriented programming languages. A subtyping relation is normally defined on a set of base types and extended to structural types by subtyping rules. Sometimes, as in Java, the set of base types does not form a lattice, that is, not any pair of base types has a join and a meet. In such a case, the type system will lose the minimal typing property when both subtyping and if-expressions are present. In other words, a term might have more than one type such that none of them is a subtype of another. To put it in another way, one can say that such a term does not have a type which

^{*}Partially supported by the NSF Grant No. CCR-9988529

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'02, October 6-8, 2002, Pittsburgh, Pennsylvania, USA.
Copyright 2002 ACM 1-58113-528-9/02/0010 ...\$5.00.

is a representative of all types of this term. For the type checking algorithms known to the author, they all work by deriving representative types of terms, therefore, they can not be used to solve the problem in this situation.

To perform a formal study of this problem, we use a minimal calculus containing subtyping and if-expressions. It is formed by adding if-expressions to λ_{\leq} , the simply typed λ -calculus with subtyping (see [Cas97]). We refer to this calculus as λ_{\leq}^{if} . The main contribution of our work is a complete and terminating type checking algorithm. Our algorithm uses only a few type annotations for a function's arguments and results, which are available in Java, C, Pascal and other imperative languages. No additional type annotation will be needed. Therefore, this technique could be used in Java to improve the subtyping without altering its syntax.

In the remainder of this section, we will explain the difficulty of type checking with subtyping and if-expressions, and then outline our approach. We also discuss a related 'subject reduction' problem, which appeared in the TYPES mailing list in June 1998 and gave the impetus to start this research.

Problem In a programming language with subtyping, a non-restricted typing rule for if-expressions could be:

$$\frac{b : Bool \quad e_1 : A \leq C \quad e_2 : B \leq C}{b?e_1:e_2 : C} \quad (1)$$

where the expression $b?e_1:e_2$ will return e_1 if b is true, or e_2 otherwise.

Type checking algorithms are normally based on the inference of a minimal (or principal) type of a term. For the if-expression $b?e_1:e_2$, such an algorithm would proceed as follows: first, it finds out the minimal upper bounds, say A and B , for e_1 and e_2 respectively; then, it derives the unique least common upper bound, say D , of A and B ; finally, it verifies that D is a subtype of C . In fact, such an algorithm was proposed in [BCM⁺93]. A requirement of this method is that every pair of types which has a common upper bound should have a common least upper bound (and also because of covariance, a similar statement with lower bounds). Under such a condition, each term has a unique minimal type. But there are cases where some types do not have a least common super type, and, as a consequence, $(b?e_1:e_2)$ might not have a unique minimal type. In those cases, no complete type checking algorithm has yet been reported in the literature.

If-expression and Subject Reduction In Java, the if-

expression $b?e_1:e_2$ allows only a restricted use of subtyping. Its typing rule can be written as:

$$\frac{b : Bool \quad e_1 : A \quad e_2 : B \quad (A \leq B) \vee (B \leq A)}{(b?e_1:e_2) : \max(A, B)} \text{ JavaIf} \quad (2)$$

where $\max(A, B)$ equals to A if $A \geq B$ or to B otherwise. Furthermore, the evaluation of a well-formed expression (containing an if-expression) might lead to a result which can not be typed by the typing rules stated in Java reference book. This later problem was pointed out in the message ‘‘Subject reduction fails for Java’’, posted by Haruo Hosoya, Benjamin Pierce and David Turner [HPT98] in TYPES mailing list in June 1998. This message provoked an active discussion among participants in the mailing list and it motivated this work.

The problem discussed in [HPT98] can be reformulated in terms of typed λ -calculus. Assume a context Γ containing the subtyping declarations $A \leq C, B \leq C$, then, we can derive the judgement:

$$\Gamma, a : A, c : B \vdash (\lambda x:C. \lambda y:C. (b?x:y))ac : C$$

where the following typing rule has been used in the typing derivation,

$$\frac{\lambda x:A.M : A \rightarrow C \quad N : B \leq C}{(\lambda x:A.M)N : C} \quad (3)$$

After β -reduction, we get $(b?a:c)$, which is not typable by rule (2).

This strange behavior has attracted interest among researchers. Here is a brief summary of some proposals¹:

The first proposal (by Sophia Drossopoulou, Donald Syme et al.) is based on the observation that the term $(b?a:c)$ in the above example can be typed by more powerful typing rule like (1). This means that such a run-time code is still typable. Therefore they propose to use two sets of typing rules: the weak one is for the source programs and it is amenable for type checking, the strong one is used to ensure that the run-time codes are always typable. This method alters neither the definition nor the implementation of Java. But the acceptable if-expressions are still limited by the weak typing rules.

The second approach (by H. Hosoya et al.) is to change the β -reduction rule so that the type information can be kept in the reduction. But it is not clear if this approach is practical in real implementations or not. Nevertheless, it is possible to add type annotations to the if-expression in the form $b?(a:A):(c:A)^2$, which ensures that the whole expression is of type A . This approach can type more terms than the first approach, but it will change the syntax of Java and requires programmers to write additional type annotations.

The third suggestion (by Tony Dekker et al.) is to extend Java so that each pair of base types (classes in Java) has a least upper bound and a greatest lower bound. Should it be possible this would be the best method. But the current version of Java does not have this property. Whether this feature could be implemented in the future or not is an open problem.

¹For more details the reader can refer to the discussion on this subject in the TYPES mailing list during June and July of 1998.

²This is pointed out by a referee of this paper.

Our approach Recall that type A is a subtype of B if any term of type A can be used in every context where a term of type B is expected. On the other hand, an expression $b?e_1:e_2$ has type C if b has type $Bool$ and both e_1 and e_2 have the same type C . These statements can be formulated as typing rules:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A \leq B}{\Gamma \vdash M : B} \text{ Subsumption}$$

$$\frac{b : Bool \quad e_1 : C \quad e_2 : C}{(b?e_1:e_2) : C} \text{ If}$$

The rule (1) can be derived by using the two rules above. Obviously, these rules are more powerful than (2) and the system will enjoy subject reduction.

The hard problem is to find a type checking algorithm. The rule *If* itself appears amenable to type checking: to check $(b?e_1:e_2) : C$, just check $b : Bool$, $e_1 : C$ and $e_2 : C$. The difficulty is the subsumption rule, which does not have the subformula property: type A in the assumption does not appear in the conclusion. One approach to get around this problem is to avoid using the subsumption rule directly, and instead, combine it with other rules. For example, the rule (3) combines application and subtyping. Similarly, we might use (1) instead of *If*. But this modification still does not solve the problem of type checking because minimal type does not always exist for an if-expression.

In order to avoid the inference of minimal types, we use complete bottom-up type checking. The main idea is to ensure that each rule has a full subformula property, which is to say that, each term and type in the assumption appears in the conclusion. An example of such a rule is the abstraction rule in lambda calculus. A counter-example is the application rule (Both rules appear among the weak typing rules in next section). To achieve our aim, the application rule is replaced by the rule *ApL*:

$$\frac{\Gamma, x : B \vdash MN_2..N_n : A \quad \Gamma \vdash N_1 : B \quad x \notin Fv(N_i)}{\Gamma \vdash (\lambda x:B.M)N_1..N_n : A} \text{ ApL}$$

The motivation of this rule comes from the observation of a typing equivalence in simply typed λ -calculus:

$$\Gamma \vdash (\lambda x:B.M)N_1..N_n : A \Leftrightarrow \Gamma \vdash (\lambda x:B.MN_2..N_n)N_1 : A$$

where $\Gamma \vdash N_1 : B \wedge x \notin Fv(N_2, \dots, N_n)$. Other rules should be modified accordingly. The new set of rules is called strong type system and is presented in Section 4.

This approach does not need the assumption that the set of base types forms a lattice. Furthermore, we will not change the syntax of Java. Type annotations are needed, but only for the arguments and results of functions that are available in most popular imperative languages such as Java, C and Pascal. The only thing remains to do is the improvement of the type checking algorithm and the modification of the typing rule. The later modification will not only simplify the typing rule but also make it more powerful. Besides, the subject reduction is valid in the proposed calculus.

Organization of the Paper The next section presents the syntax of the calculus and the weak typing rules, which is a direct extension of simply typed λ -calculus with subtyping and if-expression. In Section 3, we give the algorithms for subtyping checking. Section 4 is the main part of this

paper, where we present the strong type system, which types more terms than weak system and at the same time is a type checking algorithm. The strong typing rules are still intuitively acceptable. Besides, it enjoys good properties, including subject reduction (Section 5). All terms typable in the weak system are still typable in strong system. In other words, the strong system is complete with respect to the weak system. This result is stated at the end of Section 4 and is proved in Section 6. In the last section we summarize the contribution of this paper, discuss related works and future works.

2. THE λ_{\leq}^{if} -CALCULUS

The abstract syntax for terms and types is:

$$\begin{aligned} M &::= true \mid false \mid x \mid MM \mid \lambda x:A.M \mid (M?M:M) \\ A &::= \mathcal{B} \mid A \rightarrow A \end{aligned}$$

where \mathcal{B} denotes a base type. There is a type $Bool$ in the set of base types.

One step reduction relation $\rightarrow_{\beta if}$ is the compatible closure of

$$\begin{aligned} (\lambda x:A.M)N &\rightarrow_{\beta} M[x := N] \\ (true?M:N) &\rightarrow_{if} M \\ (false?M:N) &\rightarrow_{if} N \end{aligned}$$

The βif -reduction relation is the reflexive and transitive closure of $\rightarrow_{\beta if}$. The βif -conversion relation is the reflexive, symmetric and transitive closure of $\rightarrow_{\beta if}$. Two terms M, N are βif -convertible if the pair (M, N) belongs to the βif -conversion relation.

Let Γ be a context containing predefined typing and subtyping declarations.

$$\Gamma = \{B_1 \leq C_1, \dots, B_k \leq C_k; x_1 : A_1, \dots, x_n : A_n\}$$

where x_1, \dots, x_n are distinct variables. A_1, \dots, A_n are types, $C_1, \dots, C_k, B_1, \dots, B_k$ are base types. Note that Γ is a set, not a sequence. We use the notation $\Gamma, x : A$ to denote the context $\Gamma \cup \{x : A\}$. The set of variables in a type A is denoted by $dom(A)$. The domain of Γ , denoted by $dom(\Gamma)$, is the set $\{B_1, \dots, B_k, x_1, \dots, x_n\}$.

In the following, we use M, N, \dots to denote terms, x, x_1, \dots to denote term variables, A, B, C, \dots to denote types. ϵ denotes the empty subtyping context.

We assume that the subtyping relation defined in the context forms a preorder. The subtyping rules are fairly standard:

Subtyping rules:

$$\begin{aligned} \frac{A \leq B \in \Gamma}{\Gamma \vdash A \leq B} \text{Prim} \\ \frac{\Gamma \vdash C \leq A \quad \Gamma \vdash B \leq D}{\Gamma \vdash A \rightarrow B \leq C \rightarrow D} \rightarrow_{\leq} \\ \frac{\Gamma \vdash A \leq B \quad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C} \text{Trans} \\ \frac{}{\Gamma \vdash A \leq A} \text{Refl} \end{aligned}$$

The typing rules are formed by adding to λ_{\leq} the weak typing rules for if-expressions and for the constants $true, false$.

Weak Typing System (WTS)

$$\begin{aligned} \frac{}{\Gamma \vdash_W true : Bool} \text{True} \\ \frac{}{\Gamma \vdash_W false : Bool} \text{False} \\ \frac{x : A \in \Gamma}{\Gamma \vdash_W x : A} \text{Var} \\ \frac{\Gamma \vdash_W b : Bool \quad \Gamma \vdash_W e_1 : C \quad \Gamma \vdash_W e_2 : C}{\Gamma \vdash_W (b?e_1:e_2) : C} \text{If} \\ \frac{\Gamma, x : B \vdash_W M : A}{\Gamma \vdash_W \lambda x:B.M : B \rightarrow A} \text{Lam} \\ \frac{\Gamma \vdash_W M : B \rightarrow A \quad \Gamma \vdash_W N : B}{\Gamma \vdash_W MN : A} \text{App} \\ \frac{\Gamma \vdash_W M : A \quad \Gamma \vdash A \leq B}{\Gamma \vdash_W M : B} \text{Subsumption} \end{aligned}$$

The combination of subtyping and if-expression brings with it some unusual features. For example, without the if-expression, the system enjoys the following property:

$$M : A \wedge M : B \Rightarrow M : C \wedge C \leq A \wedge C \leq B$$

which does not hold in the presence of if-expressions. Consider the term $(b_2?a_1:a_2)$ in the **Example 4.2** (Section 4). It has two types A, B , but there is no common lower bound, G , of A, B such that $(b_2?a_1:a_2) : G$.

We put typing and subtyping declarations in one context to avoid an additional subtyping environment. The typing declaration in a context Γ is actually irrelevant to subtyping. Therefore,

$$\Gamma, x : A \vdash B \leq C \Rightarrow \Gamma \vdash B \leq C$$

3. SUBTYPING CHECKING

The subtyping checking algorithm can be formed by the standard technique. See for example the work of Curien and Ghelli for F_{\leq} [CG92]. In fact, we can construct the following algorithmic subtyping system.

$$\begin{aligned} \frac{A \leq B \in \Gamma \quad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C} \text{Prim} \\ \frac{\Gamma \vdash C \leq A \quad \Gamma \vdash B \leq D}{\Gamma \vdash A \rightarrow B \leq C \rightarrow D} \text{SApp} \\ \frac{A \text{ is a base type}}{\Gamma \vdash A \leq A} \text{ReflB} \end{aligned}$$

Note that this system does not contain the transitivity rule. Both the transitivity (Trans) rule and the general reflexivity (Refl) rule are admissible. Such a technique of achieving the transitivity elimination can be traced to the early work of Curien and Ghelli in their study of F_{\leq} [CG92].

Using the standard transitivity elimination technique, it is easy to show that this algorithmic subtyping system is equivalent to the original subtyping system.

LEMMA 3.1 (DECIDABILITY OF SUBTYPING). 1. *The algorithmic subtyping system is equivalent to the original subtyping system;*

2. *The algorithm defined by the algorithmic subtyping system terminates;*

3. *Subtyping checking is decidable.*

The proofs of these results are easy and omitted.

Since the algorithmic subtyping and original subtyping are equivalent, we will always use the notation $\Gamma \vdash A \leq B$ for a subtyping judgement.

4. STRONG TYPING AND TYPE CHECKING

Now we study the type checking problem. Recall that the type checking problem is that “Given Γ, M, A , Check $\Gamma \vdash M : A$ ”. The traditional type checking method is to derive the minimal type of M . Thus the information of the type A is not used. In contrast, we check $M : A$, but not ensure to derive the minimal type for M .

Strong Typing System (STS)

$$\frac{}{\Gamma \vdash true : Bool} \text{ True}$$

$$\frac{}{\Gamma \vdash false : Bool} \text{ False}$$

$$\frac{x : B_1 \rightarrow \dots \rightarrow B_n \rightarrow A' \in \Gamma \quad \Gamma \vdash A' \leq A \quad \Gamma \vdash N_i : B_i}{\Gamma \vdash xN_1..N_n : A} \text{ ApV}$$

$$\frac{\Gamma, x : B \vdash MN_2..N_n : A \quad \Gamma \vdash N_1 : B \quad x \notin Fv(N_i)}{\Gamma \vdash (\lambda x:B.M)N_1..N_n : A} \text{ ApL}$$

$$\frac{\Gamma, x : B \vdash M : A \quad \Gamma \vdash B' \leq B}{\Gamma \vdash \lambda x:B.M : B' \rightarrow A} \text{ ALam}$$

$$\frac{\Gamma \vdash b : Bool \quad \Gamma \vdash e_1N_1..N_n : A \quad \Gamma \vdash e_2N_1..N_n : A}{\Gamma \vdash (b?e_1:e_2)N_1..N_n : A} \text{ ApIf}$$

This set of rules can be divided into three groups. The first two rules (*True*, *False*) are typing rules for the two constants. The rule *ALam* is for λ abstraction. The remaining rules, *ApV*, *ApL* and *ApIf*, are for functional applications. In these three rules the terms in the conclusions are of the form $: MN_1..N_n$, where M can be a variable (*ApV*), a λ abstraction (*ApL*) and a if-expression (*ApIf*), respectively.

The meanings of the rules *True* and *False* are clear. The rule *ApV* states that if x is an n argument primitive function of type $B_1 \rightarrow \dots \rightarrow B_n \rightarrow A'$, A' is a subtype of A , each N_i has type B_i , then the application of x to N_1, \dots, N_n will have type A . In the special case where $n = 0$ this rule becomes:

$$\frac{\Gamma \vdash x : A' \quad \Gamma \vdash A' \leq A}{\Gamma \vdash x : A}$$

In other words, it becomes a subsumption rule for variables.

A careful reader may notice that this rule does not say that the type of each argument N_i could be a subtype of B_i . One may wonder if this type system can accept all of the terms that the previous one does. The answer to this question is Yes.

Each typing rule in the STS is syntax oriented, so this set of rules can be straightforwardly turned into a type checking algorithm, which is terminating. The subtyping checking in this algorithm will use the algorithmic subtyping rules defined in the previous section.

THEOREM 4.1 (TERMINATION). *The type checking algorithm defined by the strong type system terminates.*

PROOF. First, the subtyping checking is decidable. Second, the size of each term in a typing judgement in the assumption of each rule is smaller than that of the term in the conclusion. \square

Thus we have the decidability of type checking. On the other hand, the worst case complexity of the algorithm is exponential: due to the rule *ApIf*, the number of typing judgements appearing in the typing derivation tree is exponential to the number of embedded conditional expressions.

This system is complete with respect to the weak type system:

$$\Gamma \vdash_W M : A \Rightarrow \Gamma \vdash M : A$$

The proof of this assertion is presented in Section 6.

On the other hand, the STS is not “sound” with respect to the WTS. This is mainly due to the *ApIf* rule. The following is an example.

EXAMPLE 4.2 (A NON-TYPABLE IF-EXPRESSION). *Assume a set of base types A, B, C, D, E and a set of predefined subtyping relation:*

$$C \leq A, D \leq A, C \leq B, D \leq B$$

Note that C, D have two common upper bounds A, B , but not any least common upper bound. Assume $b_1, b_2 : Bool, e_1 : A \rightarrow E, e_2 : B \rightarrow E, a_1 : C, a_2 : D$. Then the following typing is derivable in the weak type system:

$$(b_1?(e_1(b_2?a_1:a_2)) : (e_1(b_2?a_1:a_2))) : E$$

The following term is β_{if} -convertible to the above term, but it is typable only in the STS, not in the WTS:

$$(b_1?e_1:e_2)(b_2?a_1:a_2)$$

Observe that $(b_2?a_1:a_2)$ has two types: A and B , $(b_1?e_1:e_2)$ has two types $C \rightarrow E$ and $D \rightarrow E$, which are the only common upper bounds of $A \rightarrow E$ and $B \rightarrow E$. Neither A nor B is a subtype of C or D .

This example shows that this set of rules is more powerful than the set of WTS. That is, STS can type more terms than WTS does. This is justified since each rule in STS has intuitively reasonable interpretation. For example, the rule *ApL* says that if, under the environment $x : B$, M is a function applicable to N_2, \dots, N_n and the application has the type A , then $\lambda x:B.M$ should be applicable to N_1, \dots, N_n where N_1 is of type B . On the other hand, we show that this system has good properties, including admissibilities of subsumption and application and subject reduction (see Section 5). Therefore, we recommend using STS in practice.

5. SUBJECT REDUCTION

Before proving the main results, we present a set of structural lemmas. Because their proofs are standard, so they are omitted. At the end of this section, we show the subject reduction and a key lemma.

LEMMA 5.1 (NARROWING). $\Gamma, x : A \vdash M : B \wedge \Gamma \vdash A' \leq A \Rightarrow \Gamma, x : A' \vdash M : B$

LEMMA 5.2 (WEAKENING). $\Gamma \vdash M : B \wedge x \notin \Gamma \Rightarrow \Gamma, x : A \vdash M : B$

LEMMA 5.3 (GENERATION OF SUBTYPING). $\Gamma \vdash A \rightarrow B \leq C \rightarrow D \Rightarrow \Gamma \vdash C \leq A \wedge \Gamma \vdash B \leq D$

LEMMA 5.4 (GENERATION OF TYPING).

$$\begin{aligned} & \Gamma \vdash x : A \\ \Rightarrow & x : B \in \Gamma \wedge \Gamma \vdash B \leq A \\ & \Gamma \vdash \lambda x : B. M : A \\ \Rightarrow & A \equiv C \rightarrow D \wedge \Gamma \vdash C \leq B \wedge \Gamma, x : B \vdash M : D \\ & \Gamma \vdash MN : A \wedge M \neq (b?e_1:e_2)N_1..N_n \\ \Rightarrow & \exists B. \text{ s.t. } \Gamma \vdash M : B \rightarrow A \wedge \Gamma \vdash N : B \\ & \Gamma \vdash (b?e_1:e_2)N_1..N_n : A \\ \Rightarrow & \Gamma \vdash e_1N_1..N_n : A \wedge \Gamma \vdash e_2N_1..N_n : A \wedge \Gamma \vdash b : Bool \end{aligned}$$

where $n \geq 0$.

Note that the condition $M \neq (b?e_1:e_2)N_1..N_n$ is necessary in the third property. The assertion

$$\Gamma \vdash MN : A \Rightarrow \exists B. \text{ s.t. } \Gamma \vdash M : B \rightarrow A \wedge \Gamma \vdash N : B$$

is not valid in general. A counterexample is the term $(b_1?e_1:e_2)(b_2?a_1:a_2)$ in the example from the previous section.

LEMMA 5.5 (UNICITY OF TYPE ARITY).

1. $\Gamma \vdash C \leq A_1 \rightarrow \dots \rightarrow A_n \Rightarrow C \equiv B_1 \rightarrow \dots \rightarrow B_n$
2. $\Gamma \vdash A_1 \rightarrow \dots \rightarrow A_n \leq C \Rightarrow C \equiv B_1 \rightarrow \dots \rightarrow B_n$
3. $\Gamma \vdash M : A_1 \rightarrow \dots \rightarrow A_n \rightarrow A$
 $\wedge \Gamma \vdash M : B_1 \rightarrow \dots \rightarrow B_m \rightarrow A \Rightarrow n = m$

PROOF. 1., 2. Induction on the structure of C ; 3. induction on the structure of M . \square

LEMMA 5.6.

$$\begin{aligned} & \Gamma \vdash (\lambda x : B. M)N_1..N_n : A \\ \Rightarrow & \Gamma \vdash (\lambda x : B. MN_2..N_n)N_1 : A \wedge \Gamma \vdash N_1 : B \end{aligned}$$

PROOF. The assertion is proved by using the “generation of typing” lemma and the typing rules.

(\Rightarrow).

$$\begin{aligned} & \Gamma \vdash (\lambda x : B. M)N_1..N_n : A \\ \Rightarrow & \Gamma \vdash (\lambda x : B. M)N_1..N_{n-1} : B_n \rightarrow A \\ & \wedge \Gamma \vdash N_n : B_n \wedge x \notin Fv(N_n) \\ & \dots \dots \\ \Rightarrow & \Gamma \vdash \lambda x : B. M : B_1 \rightarrow \dots \rightarrow B_n \rightarrow A \\ & \wedge \Gamma \vdash N_i : B_i \wedge x \notin Fv(N_i) \quad i = 1..n \\ \Rightarrow & \Gamma, x : B \vdash M : B_2 \rightarrow \dots \rightarrow B_n \rightarrow A \\ & \wedge \Gamma \vdash B_1 \leq B \wedge \Gamma \vdash N_i : B_i \quad i = 1..n \\ \Rightarrow & \Gamma, x : B \vdash MN_2..N_n : A \wedge \Gamma \vdash B_1 \leq B \\ & \wedge \Gamma \vdash N_1 : B_1 \\ \Rightarrow & \Gamma \vdash (\lambda x : B. MN_2..N_n) : B \rightarrow A \wedge \Gamma \vdash N_1 : B \\ \Rightarrow & \Gamma \vdash (\lambda x : B. MN_2..N_n)N_1 : A \end{aligned}$$

\square

Due to the if-expression $(b?e_1:e_2)$, the reverse of this lemma is not always valid. Consider the judgement

$$\Gamma \vdash (\lambda x : B. (b?e_1:e_2)N_2)N_1 : A$$

and assume $x \notin Fv(N_2)$. The derivability of this judgement implies that $\Gamma, x : B \vdash e_i N_2 : A$ for $i = 1, 2$. But this does not mean that the if-expression $b?e_1:e_2$ itself is typable. Therefore, we do not know if the judgement $\Gamma \vdash (\lambda x : B. (b?e_1:e_2))N_1 N_2 : A$ is derivable or not.

LEMMA 5.7 (UNIQUE DERIVATION). *In STS, a typable term has exactly one derivation.*

PROOF. Just notice that the last rule in the derivation of a term is uniquely determined. \square

LEMMA 5.8 (ADMISSIBILITY OF *Var* AND *Lam*). *The rule *Var* and the rule *Lam* in WTS are admissible in STS.*

PROOF. Observe that the rule *Var* in WTS is a special case of the rule *ApV* in STS (with $n = 0$); and that the rule *Lam* in WTS is a special case of the rule *ApL* in STS (with $n = 1$).

\square

Now, we prove that both the subsumption rule and the application rule are admissible in the STS. The proofs are standard induction on a main term M in the statement. Since each term has a unique typing derivation by the STS, the case analysis in this proof is closely related to the strong typing derivation.

LEMMA 5.9 (ADMISSIBILITY OF SUBSUMPTION).

$$\Gamma \vdash M : A \wedge \Gamma \vdash A \leq B \Rightarrow \Gamma \vdash M : B$$

PROOF. Induction on the size of M . M can be written in the form of $PN_1..N_n$ where P is not an application. The proof proceeds by analyzing the form of P .

We treat only the case where $M = \lambda y : B'. M'$.

$$\begin{aligned} & \Gamma \vdash \lambda y : B'. M' : A_1 \rightarrow B_1 \\ \Rightarrow & \Gamma, y : B' \vdash M' : B_1 \wedge \Gamma \vdash A_1 \leq B' \quad ALam \end{aligned}$$

$$\begin{aligned} & \Gamma \vdash A_1 \rightarrow B_1 \leq A_2 \rightarrow B_2 \\ \Rightarrow & \Gamma \vdash A_2 \leq A_1 \wedge \Gamma \vdash B_1 \leq B_2 \\ \Rightarrow & \Gamma, y : B' \vdash M' : B_2 \wedge \Gamma \vdash A_2 \leq B' \quad IH \\ \Rightarrow & \Gamma \vdash \lambda y : B'. M' : A_2 \rightarrow B_2 \quad ApL \end{aligned}$$

\square

LEMMA 5.10 (ADMISSIBILITY OF APPLICATION).

$$\Gamma \vdash M : A \rightarrow B \wedge \Gamma \vdash N : A \Rightarrow \Gamma \vdash MN : B$$

PROOF. Induction on the size of M . M can be written in the form of $PN_1..N_n$ where P is not an application. The proof proceeds by analyzing the form of P .

We treat only the case where $M = \lambda y : B'. M'$.

$$\begin{aligned} & \Gamma \vdash \lambda y : B'. M' : A \rightarrow B \wedge \Gamma \vdash N : A \quad \text{assumption} \\ \Rightarrow & \Gamma, y : B' \vdash M' : B \wedge \Gamma \vdash A \leq B' \quad ALam \\ \Rightarrow & \Gamma \vdash N : B' \quad \text{Lemma 5.9} \\ \Rightarrow & \Gamma \vdash (\lambda y : B'. M')N : B \quad ApL \end{aligned}$$

\square

LEMMA 5.11 (SUBSTITUTION).

$$\Gamma, x : B \vdash M : A \wedge \Gamma \vdash N : B \Rightarrow \Gamma \vdash M[x := N] : A$$

PROOF. Induction on the size of M . M can be written in the form of $PN_1..N_n$ where P is not an application. The proof proceeds by analyzing the form of P .

We treat only the case where $M = xN_1..N_n$.

$$\begin{aligned} & \Gamma, x : B \vdash xN_1..N_n : A && \text{assumption} \\ \Rightarrow & B \equiv B_1 \rightarrow \dots \rightarrow B_n \rightarrow A' \wedge \Gamma \vdash A' \leq A && \\ & \wedge \Gamma, x : B \vdash N_i : B_i \text{ where } i = 1..n && \text{ApV} \\ \Rightarrow & \Gamma \vdash N_i[x := N] : B_i \quad i = 1..n && \text{IH} \\ \Rightarrow & \Gamma \vdash NN_1[x := N]..N_n[x := N] : A && \text{Lemma 5.10} \\ \Rightarrow & \Gamma \vdash M[x := N] : A && \end{aligned}$$

□

The main result of this section is the subject reduction, which states that the type of a well-formed term will be preserved in the reduction.

THEOREM 5.12 (SUBJECT REDUCTION).

$$\Gamma \vdash M : A \wedge M \rightarrow_{\beta} M' \Rightarrow \Gamma \vdash M' : A$$

PROOF. Induction on the derivation of $\Gamma \vdash M : A$. We treat the case where the last rule used in the derivation is ApL :

$$\frac{\Gamma, x : B \vdash MN_2..N_n : A \quad \Gamma \vdash N_1 : B \quad x \notin Fv(N_i)}{\Gamma \vdash (\lambda x:B.M)N_1..N_n : A} \text{ApL}$$

There are three cases.

Case $N_1 \rightarrow_{\beta} N'_1$. By induction hypothesis, $\Gamma \vdash N'_1 : B$, the result follows.

Case $M \rightarrow_{\beta} M'$ or $N_i \rightarrow_{\beta} N'_i \quad i = 2..n$. Assume $M \rightarrow_{\beta} M'$, then by induction hypothesis, $\Gamma, x : B \vdash M'N_2..N_n : A$, the result follows. The cases for $N_i \rightarrow_{\beta} N'_i \quad i = 2..n$ are similar.

Case $(\lambda x:B.M)N_1..N_n \rightarrow_{\beta} M[x := N_1]N_2..N_n$. Since $x \notin Fv(N_2, \dots, N_n)$, we have

$$\begin{aligned} & \Gamma, x : B \vdash MN_2..N_n : A \wedge \Gamma \vdash N_1 : B \\ \Rightarrow & \Gamma \vdash M[x := N_1]N_2..N_n : A && \text{Lemma 5.11} \end{aligned}$$

□

6. COMPLETENESS OF STRONG TYPING

In this section, we prove that terms typable in the WTS are all typable in the STS. First, we show that the use of subsumption in the WTS can be kept to minimum. This technique of subsumption elimination is adapted from [Cas97]. This is one of the essential steps for the proof of completeness. More precisely, we show that the subsumption rule (along with $Var+$ and Lam rule) can be replaced by the rules $Var+$ and $ALam$ in the following intermediate type system. $Var+$ combines subsumption with the typing of the term variable, $ALam$ can be viewed as the subsumption rule for λ -abstraction.

Intermediate Typing System (ITS)

$$\begin{aligned} & \frac{}{\Gamma \vdash_I true : Bool} \text{True} \\ & \frac{}{\Gamma \vdash_I false : Bool} \text{False} \\ & \frac{x : A \in \Gamma \quad \Gamma \vdash A \leq B}{\Gamma \vdash_I x : B} \text{Var+} \\ & \frac{\Gamma, x : B \vdash_I M : A \quad \Gamma \vdash C \leq B}{\Gamma \vdash_I \lambda x:B.M : C \rightarrow A} \text{ALam} \\ & \frac{\Gamma \vdash_I M : B \rightarrow A \quad \Gamma \vdash_I N : B \quad M \not\equiv (b?e_1:e_2)N_1..N_n}{\Gamma \vdash_I MN : A} \text{App} \\ & \frac{\Gamma \vdash_I b : Bool \quad \Gamma \vdash_I e_1N_1..N_n : A \quad \Gamma \vdash_I e_2N_1..N_n : A}{\Gamma \vdash_I (b?e_1:e_2)N_1..N_n : A} \text{ApIf} \end{aligned}$$

LEMMA 6.1 (ELIMINATION OF SUBSUMPTION). *The WTS is equivalent to the ITS.*

PROOF. First note that the new rules $Var+$ and $ALam$ are admissible in the WTS. So we need only to show that each typing judgement derivable in the WTS can be derived in the ITS. The proof proceeds by induction on the typing derivation $\Gamma \vdash_W M : A$. Analyze according to the form of M . The result follows from the following transformations on typing derivation. We use the notation $\Gamma \vdash_W M : A \leq B$ as a shorthand for the conjunction $\Gamma \vdash_W M : A$ and $\Gamma \vdash_W A \leq B$. Similarly, we use the notation $\Gamma \vdash_I M : A \leq B$.

$$\begin{aligned} & \frac{\Gamma \vdash_W M : A \rightarrow B \quad \Gamma \vdash_W N : A}{\Gamma \vdash_W MN : B \leq C} \\ & \frac{}{\Gamma \vdash_W MN : C} \\ & \frac{\Gamma \vdash_I M : A \rightarrow B \leq A \rightarrow C}{\Gamma \vdash_I M : A \rightarrow C} \quad \Gamma \vdash_I N : A \\ & \frac{}{\Gamma \vdash_I MN : C} \end{aligned}$$

$$\begin{aligned} & \frac{\Gamma, x : A \vdash_W M : B}{\Gamma \vdash_W \lambda x:A.M : A \rightarrow B \leq C \rightarrow D} \\ & \frac{}{\Gamma \vdash_W \lambda x:A.M : C \rightarrow D} \\ & \frac{\Gamma, x : A \vdash_I M : B \leq D}{\Gamma, x : A \vdash_I M : D} \quad \Gamma \vdash C \leq A \\ & \frac{}{\Gamma \vdash_I \lambda x:A.M : C \rightarrow D} \end{aligned}$$

$$\begin{aligned} & \frac{\Gamma \vdash_W b : Bool \quad \Gamma \vdash_W e_1 : D \quad \Gamma \vdash_W e_2 : D}{\Gamma \vdash_W (b?e_1:e_2) : D \leq C} \\ & \frac{}{\Gamma \vdash_W (b?e_1:e_2) : C} \\ & \frac{\Gamma \vdash_I e_1 : D \leq C \quad \Gamma \vdash_I e_2 : D \leq C}{\Gamma \vdash_I e_1 : C \quad \Gamma \vdash_I e_2 : C} \\ & \frac{}{\Gamma \vdash_I (b?e_1:e_2) : C} \end{aligned}$$

□

THEOREM 6.2 (COMPLETENESS OF STRONG TYPING). *Any judgement derivable in the ITS is derivable in the STS. Therefore, any judgement derivable in the WTS is derivable in the STS.*

PROOF. Induction on the size of the term in the ITS typing judgement. We need only to consider the case where the last step is derived by the rules $Var+$ or App . The $Var+$ rule is the special case of the ApV rule by setting $n = 0$. For the App rule, assume the last step of the typing derivation is:

$$\frac{\Gamma \vdash_I M : B \rightarrow A \quad \Gamma \vdash_I N : B \quad M \neq (b?e_1:e_2)N_1..N_n}{\Gamma \vdash_I MN : A} \text{App}$$

M can have only two possible forms: $xN_1..N_n$ and $(\lambda x:C.P)N_1..N_n$.

Case $M \equiv xN_1..N_n$. Assume $x : C \in \Gamma$,

$$\begin{aligned} & \Gamma \vdash_I xN_1..N_n : B \rightarrow A \\ \Rightarrow & \Gamma \vdash_I x : B_1 \rightarrow \dots \rightarrow B_n \rightarrow B \rightarrow A \\ & \wedge \Gamma \vdash_I N_i : B_i \quad i = 1..n \end{aligned}$$

$$\begin{aligned} & x : C \in \Gamma \\ \Rightarrow & \Gamma \vdash C \leq B_1 \rightarrow \dots \rightarrow B_n \rightarrow B \rightarrow A \\ \Rightarrow & C \equiv B'_1 \rightarrow \dots \rightarrow B'_n \rightarrow B' \rightarrow A' \\ \Rightarrow & \Gamma \vdash A' \leq A \wedge \Gamma \vdash B \leq B' \\ & \wedge \Gamma \vdash B_i \leq B'_i \quad i = 1..n \\ \Rightarrow & \Gamma \vdash_I N_i : B'_i \quad i = 1..n \\ \Rightarrow & \Gamma \vdash N_i : B_i \quad i = 1..n \end{aligned} \quad IH$$

$$\begin{aligned} & \Gamma \vdash_I N : B \\ \Rightarrow & \Gamma \vdash_I N : B' \\ \Rightarrow & \Gamma \vdash N : B' \\ \Rightarrow & \Gamma \vdash xN_1..N_n N : A \end{aligned} \quad \begin{array}{l} IH \\ ApV \end{array}$$

Case $M \equiv (\lambda x:C.P)N_1..N_n$.

$$\begin{aligned} & \Gamma \vdash_I (\lambda x:C.P)N_1..N_n : B \rightarrow A \wedge \Gamma \vdash_I N : B \\ \Rightarrow & \Gamma \vdash_I (\lambda x:C.P) : B_1 \rightarrow \dots \rightarrow B_n \rightarrow B \rightarrow A \\ & \wedge \Gamma \vdash_I N_i : B_i \quad i = 1..n \\ \Rightarrow & \Gamma, x : C \vdash_I P : B_2 \rightarrow \dots \rightarrow B_n \rightarrow B \rightarrow A \\ & \wedge \Gamma \vdash B_1 \leq C \\ \Rightarrow & \Gamma, x : C \vdash_I PN_2..N_n N : A \wedge \Gamma \vdash_I N_1 : C \\ \Rightarrow & \Gamma, x : C \vdash PN_2..N_n N : A \wedge \Gamma \vdash N_1 : C \\ \Rightarrow & \Gamma \vdash (\lambda x:C.P)N_1..N_n N : A \end{aligned} \quad \begin{array}{l} IH \\ ApL \end{array}$$

□

7. CONCLUDING REMARKS

Summary In this paper, an extension of simply typed λ -calculus with both if-expression and subtyping is presented. The challenging aspect of this research is that the calculus does not have the minimal typing property, because we do not assume that the base types form a lattice.

We propose a weak type system (WTS) and a strong type system (STS). Both systems are more general than the typing rules used in Java for type checking if-expression. The WTS is close to the traditional presentation of type system, and is intuitively understandable. The STS types more terms than the WTS and it is an algorithmic typing system. The type checking for the STS is decidable. The STS is shown to be complete with respect to the WTS. We have proved that the STS enjoys subject reduction.

This work would be particularly useful for a full implementation of subtyping in imperative object oriented languages.

From the methodological point of view, our type checking algorithm does not belong to the traditional type checking technique which is based on the derivation of representative

types (principal types, minimal types etc.). Therefore this method might be adapted to other situations where traditional type checking technique fails.

Related Works In [BCM⁺93], Kim Bruce et al. proposed a type checking algorithm for a calculus with subtyping and if-expression. But they have assumed that the set of base types forms a lattice, so that terms in their calculus have minimal types. Our work does not need this assumption.

Various works exist on the subject of polymorphic subtyping. Among these is the Henglein's [Hen96]. In his system, it is possible to introduce a constant *if* with type scheme $\forall \alpha. (Bool \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$. Henglein has proposed a sound and complete type checking algorithm which, given a typing environment Γ and a term e , derives a principal typing judgement of the form $C, \Gamma \vdash e : \tau$, where C is a set of additional subtyping assumptions (called constraints). The typability of e depends on whether the set of constraints C is solvable or not. In our approach, we check $\Gamma \vdash e : \tau$ directly without the need of the constraints C .

Future Works This work represents one step toward resolving the problem of combining subtyping and if-expression. More work remains to be done. One direction is to extend this technique to allow the full use of subsumption in Java. Another is to extend this technique to add if-expression to F_{\leq} [CG92], which is a second order λ -calculus with bounded quantification and is taken as a formal model for object-oriented language.

Acknowledgment This study is inspired by the message "subject reduction fails in Java" posted by Haruo Hosoya, Benjamin Pierce and David Turner in types mailing list. Later discussions around this message by Kim Bruce, Hosoya et al and others on the types mailing list have clarified the problem. The author would like to thank John Maraist, Martin Odersky, Hongwei Xi and Christophe Zenger for helpful discussions. Thanks go to anonymous referees for their remarks. I am grateful to Paul Johnson at Berlitz for his help with my English.

8. REFERENCES

- Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type-checking in an object-oriented language. In *OOPSLA '93*, 1993.
- G. Chen. Subtyping and if-expression. Technical Report, School of Computer and Information Science. University of South Australia, 1999.
- G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkäuser, Boston, 1997. ISBN 3-7643-3905-5.
- P. L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and the type checking in F_{\leq} . *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.
- J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.
- F. Henglein. Syntactic properties of

polymorphic subtyping. TOPPS Technical Report (D-report series) D-293, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, May 1996.
H. Hosoya, B. Pierce, and D. Turner. subject reduction fails in java, June 1998. Message on Types electronic mailing list.