*Int J Software Informatics*, Volume 10, Issue 4 (2016), pp. 000–000
International Journal of Software and Informatics, ISSN 1673-7288

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661048

DOI: 10.21655/ijsi.1673-7288.00236

# Second Order Bounded Quantification with If-Expression

Gang Chen

(Beijing Jinghang Research Institute of Computing and Communication, Beijing 100074, China)

**Abstract** The full combination of if-expression and subtyping was known as a challenging problem for a long time. The difficulty comes from the fact that two branches of an if-expression might have different types and these types might not have a unique least upper bound. Such situation could happen in real object oriented languages such as Java. In our previous work, we investigated this problem in a simply typed lambda calculus extended with subtyping and if-expression and solved the type checking problem in that system. In this paper, we extend the result to bounded quantification combined with if-expression.
**Key words:** subtyping; if-expression; bounded quantification; type theory; programming language

**Chen G. Second order bounded quantification with if-expression.** *Int J Software Informatics*, **Vol.10, No.4 (2016): 000–000. http://www.ijsi.org/1673-7288/10/236.htm**

## 1 Introduction

Cardelli and Wegner[10] have introduced the notion of bounded quantification to combine typed polymorphic $\lambda$-calculus with subtyping. Since then, this feature has been extensively studied by many researchers, for example Refs. [2,3,6,9,8,12]. The most well known system with bounded quantification is the system $F_{\leqslant}$[1][6], which extends the system-F with second order bounded quantification.

The system $F_{\leqslant}$ serves as a model to represent polymorphic and object-oriented features in programming languages. In Ref. [1], Bruce, et al. proposed an object oriented language TOOPLE based on $F_{\leqslant}$. In their work, the difficulty of combining subtyping and if-expression is revealed. At the presence of if-expression, an expression might not have a minimal type. As a consequence, the traditional type checking technique based on minimal type inference does not work. Bruce, et al.[1] managed to get around the problem by adding the assumption that the set of base types forms a lattice. Under this condition, it is provable that the set of types also forms a lattice. This fact ensures the existence of minimal type for every pair of types.

Unfortunately, this assumption is not valid in all situations. One example is Java. In fact, Java does not utilize the full power of subtyping, instead it only has a fairly restricted typing rule for if-expression, in which the type of one branch must be

Corresponding author: Gang Chen, Email: gangchensh@qq.com
Received 2016-07-31; Revised 2016-12-12; Accepted 2017-01-07.

[1]The subtyping relation $\leqslant$ is sometimes written as $<:$ in the literature.

a subtype of another. Although this approach simplifies Java type checking, but, in 1998, Hosoya, et al. pointed out (in their message to Types mailing list) that, with this if-expression typing rule, subject reduction will fail. Since then, several proposals have been raised to solve or to explain the problem (see also a discussion in Ref. [7]), but none of them provides a complete type checking algorithm for calculus with full subsumption and if-expression.

In Ref. [7], we investigated this problem in a calculus $\lambda_{\leqslant}^{if}$, which extends the simply typed $\lambda$-calculus with subtyping and if-expression. In this calculus, the set of base types need not form a lattice, so expressions might not have minimal types. In order to develop a type checking algorithm, we reformulated the type system so that it corresponds directly to an algorithm. The resulted system is called Strong Typing System (STS) because it actually types more terms than the intuitively formed type system which is also called Weak Typing System (WTS). The type checking problem for STS is proved to be decidable. Besides, this system enjoys the subject reduction property.

The main purpose of this paper is to extend this result to $F_{\leqslant}$. We will study a calculus $F_{\leqslant}^{if}$, which extends the kernel $F_{\leqslant}{}^{[13]}$ with if-expression. It is well known that the type checking problem for the full $F_{\leqslant}$ is undecidable, but this problem is decidable for the kernel $F_{\leqslant}{}^{[13]}$. As in our previous work, we will first present an intuitively acceptable type system, which will be called Weak Typing System (WTS). Later, a Strong Typing System (STS) will be formulated. STS types more terms than WTS and the type checking problem for STS is decidable.

Next section explains the difficulty of type checking with subtyping and if-expressions, as well as the related 'subject reduction' problem, which appeared in the TYPES mailing list in June 1998 and gave the impetus to start this research. Then, we will discuss the related works and outline our approach. In Section 3, we review the system $\lambda_{\leqslant}^{if}$, which combines subtyping and if-expression in simply typed $\lambda$-calculus. The system in this paper is the generalization of $\lambda_{\leqslant}^{if}$. In Section 4, we present the syntax, typing and subtyping rules of the calculus $F_{\leqslant}^{if}$. The type system is also named as Weak Typing System (WTS). Section 5 discusses the subtyping checking problem, which is essentially same as $F_{\leqslant}$ and is known to be decidable. The Strong Typing System is formulated in Section 6, where an example is given to show that there are expressions typable is STS but not in WTS. The STS is syntax directed[2] and satisfies the *subformula property*[3]. Therefore STS specify a type checking algorithm. Section 7.2 shows that this algorithm terminates. Hence the type checking problem for STS is decidable. In Section 8, we prove that typing rules of WTS are all admissible in STS. Therefore, any term typable in WTS is also typable in STS. This shows that STS is strictly stronger than WTS. The subject reduction property is proved in Section 9. The last section is the summary of this work. Structural properties are left in the Appendix.

## 2 Motivation and Related Work

[2]A set of rules is *syntax directed* iff there is a one-to-one correspondence between provable judgments and proof trees.

[3]A rule satisfies *subformula property* iff all the formulae appearing in the premises of the rule are subformulae of those appearing in the conclusion.

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

$$\frac{b : \mathbf{bool} \quad e_1 : A \leqslant C \quad e_2 : B \leqslant C}{b?e_1{:}e_2 : C} \tag{1}$$

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 Ref. [1]. 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:

$$\frac{b : \mathbf{bool} \quad e_1 : A \quad e_2 : B \quad (A \leqslant B) \vee (B \leqslant A)}{(b?e_1{:}e_2) : max(A, B)}\ JavaIf \tag{2}$$

where $\max(A, B)$ equals to $A$ if $A \geqslant B$ or to $B$ otherwise. Furthermore, the evaluation of a well-formed expression (containing a 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[11] 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 Ref. [11] can be reformulated in terms of typed $\lambda$-calculus. Assume a context $\Gamma$ containing the subtyping declarations $A \leqslant C, B \leqslant C$, where neither A or B is a subtype of another, and the typing declarations $b : \mathbf{bool}, a : A, c : B$, then, we can have the following typing derivation using JavaIf, abstraction, application and the subsumption rules:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma{\vdash}b{:}\mathbf{bool} \quad \Gamma, x{:}C, y{:}C{\vdash}x{:}C \quad \Gamma, x{:}C, y{:}C{\vdash}y{:}C}{\Gamma, x{:}C, y{:}C{\vdash}b?x{:}y{:}C}}{\Gamma, x{:}C{\vdash}\lambda y{:}C.(b?x{:}y){:}C{\to}C}}{\Gamma{\vdash}(\lambda x{:}C.\lambda y{:}C.(b?x{:}y)){:}C{\to}(C{\to}C)} \qquad \dfrac{\Gamma{\vdash}a{:}A \quad \Gamma{\vdash}A{\leqslant}C}{\Gamma{\vdash}a{:}C}}{\Gamma{\vdash}(\lambda x{:}C.\lambda y{:}C.(b?x{:}y))a{:}C{\to}C} \qquad \dfrac{\Gamma{\vdash}c{:}B \quad \Gamma{\vdash}B{\leqslant}C}{\Gamma{\vdash}c{:}C}}{\Gamma{\vdash}(\lambda x{:}C.\lambda y{:}C.(b?x{:}y))ac{:}C} \tag{3}$$

After $\beta$-reduction over the last term in the conclusion, 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 proposals[4]:

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 $\beta$-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 $b?(a{:}A){:}(c{:}A)$[5], 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 extra type annotation in each branch of an if-expression.

The third suggestion (by Tony Dekker et al.) is to extend the type system in such a way that each pair of base types has a least upper bound and a greatest lower bound, that is, the set of types form a lattice. As a matter of fact, the system proposed by Bruce et al[1] has been made to have this property. However, "computing minimum types for conditional statements was unexpectedly complex"[1].

This paper presents an algorithmic type system, which 1) is not as complex as the first one, and 2) does not need extra type annotation on branches of if-expression as the second approach, but still admits same set of terms, 3) it does not have the restriction that the set of types have to form a lattice as required by the third proposal, and type checking procedure is simpler and more efficient.

**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:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A \leqslant B}{\Gamma \vdash M : B}\ \textit{Subsumption}$$

$$\frac{b : \mathbf{bool} \quad e_1 : C \quad e_2 : C}{(b?e_1{:}e_2) : C}\ \textit{If}$$

The rule (1) can be derived by using the two rules above. 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 $b : \mathbf{bool}$, $e_1 : C$ 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,

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

[5]This is pointed out by a referee of this paper.

combine it with other rules. 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*:

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

The motivation of this rule comes from the observation of a typing equivalence in simply typed $\lambda$-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 $\Gamma \vdash N_1 : B \wedge\ x \notin Fv(N_2, .., N_n)$. Other rules should be modified accordingly. The new set of rules is called strong type system and is presented in Section 6.

This approach does not need the assumption that the set of base types forms a lattice. Furthermore, it does not require adding type annotations on each branch of if-expression. 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.

## 3 The $\lambda^{if}_{\leqslant}$-calculus

This section briefly review the $\lambda^{if}_{\leqslant}$-calculus and its main results. Details about this calculus can be found in Ref. [7].

The abstract syntax for terms and types of $\lambda^{if}_{\leqslant}$ is:

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

where $\mathcal{B}$ is a nonempty set of constant types, also called base types, and there is a type **bool** in $\mathcal{B}$.

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

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

The $\beta if$-reduction relation is the reflexive and transitive closure of $\to_{\beta if}$. The $\beta_{if}$-conversion relation is the reflexive, symmetric and transitive closure of $\to_{\beta if}$.

Two terms $M, N$ are $\beta_{if}$-convertible if the pair $(M, N)$ belongs to the $\beta_{if}$-conversion relation.

Let $\Gamma$ be a context containing predefined typing and subtyping declarations.

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

where $x_1, ..x_n$ are distinct variables. $A_1, .., A_n$ are types, $C_1, ..C_k, B_1, .., B_k$ are base types. Note that $\Gamma$ 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 $\Gamma$, denoted by $dom(\Gamma)$, is the set $\{B_1, \cdots, B_k, x_1, \cdots, x_n\}$.

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

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

$\lambda^{if}_{\leqslant}$ **Subtyping rules:**

$$\frac{A \leqslant B \in \Gamma}{\Gamma \vdash A \leqslant B}\ Prim$$

$$\frac{\Gamma \vdash C \leqslant A \quad \Gamma \vdash B \leqslant D}{\Gamma \vdash A{\to}B \leqslant C{\to}D}\ {\to}_{\leqslant}$$

$$\frac{\Gamma \vdash A \leqslant B \quad \Gamma \vdash B \leqslant C}{\Gamma \vdash A \leqslant C}\ Trans$$

$$\frac{}{\Gamma \vdash A \leqslant A}\ Refl$$

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

$\lambda^{if}_{\leqslant}$ **Weak Typing System ($\lambda^{if}_{\leqslant}$-WTS)**

$$\frac{}{\Gamma \vdash_W true : \mathbf{bool}}\ True$$

$$\frac{}{\Gamma \vdash_W false : \mathbf{bool}}\ False$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash_W x : A}\ Var$$

$$\frac{\Gamma \vdash_W b : \mathbf{bool} \quad \Gamma \vdash_W e_1 : C \quad \Gamma \vdash_W e_2 : C}{\Gamma \vdash_W (b?e_1{:}e_2) : C}\ If$$

$$\frac{\Gamma, x : B \vdash_W M : A}{\Gamma \vdash_W \lambda x{:}B.M : B{\to}A}\ Lam$$

$$\frac{\Gamma \vdash_W M : B{\to}A \quad \Gamma \vdash_W N : B}{\Gamma \vdash_W MN : A}\ App$$

$$\frac{\Gamma \vdash_W M : A \quad \Gamma \vdash A \leqslant B}{\Gamma \vdash_W M : B}\ Subsumption$$

For the purpose of subtyping checking and type checking, we have developed algorithmic subtyping system:

$$\frac{A \leqslant B \in \Gamma \quad \Gamma \vdash B \leqslant C}{\Gamma \vdash A \leqslant C}\ Prim$$

$$\frac{\Gamma \vdash C \leqslant A \quad \Gamma \vdash B \leqslant D}{\Gamma \vdash A{\to}B \leqslant C{\to}D}\ SApp$$

$$\frac{A \textit{ is a base type}}{\Gamma \vdash A \leqslant A}\ ReflB$$

and the $\lambda^{if}_{\leqslant}$ Strong Typing System:

$$\frac{}{\Gamma \vdash \mathit{true} : \mathbf{bool}}\ True$$

$$\frac{}{\Gamma \vdash \mathit{false} : \mathbf{bool}}\ False$$

$$\frac{x : B_1{\to}\cdots{\to}B_n{\to}A' \in \Gamma \quad \Gamma \vdash A' \leqslant A \quad \Gamma \vdash N_i : B_i}{\Gamma \vdash xN_1..N_n : A}\ ApV$$

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

$$\frac{\Gamma, x : B \vdash M : A \quad \Gamma \vdash B' \leqslant B}{\Gamma \vdash \lambda x{:}B.M : B' \to A}\ ALam$$

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

It is easy to see that these systems can be turned into subtyping checking and type checking algorithms. The algorithmic subtyping system is equivalent to the subtyping system. The $\lambda^{if}_{\leqslant}$ Strong Typing System types more terms than $\lambda^{if}_{\leqslant}$ type system. Besides, subject reduction is valid for $\lambda^{if}_{\leqslant}$ Strong Typing System.

In the following, we extend $\lambda^{if}_{\leqslant}$ to a new system $F^{if}_{\leqslant}$, which combines second order bounded quantification and if-expression. The tricky part of this extension is to prove the decidability of subtyping. It requires to show that the type checking algorithm is terminating. To this purpose, we use a measure based on the $\beta_2\Gamma$-reduction, see Section 7.2.

## 4 The $F^{if}_{\leqslant}$-calculus

We use lowercase letters $s, t, a, b, e...$ to range over terms, $x, y, ...$ to range over term variables, $X, Y, ..$ to range over type variables, $A, B, C, ..$ to range over types and $M, N, ..$ to range over expressions.

The types, terms, expressions and contexts of $F^{if}_{\leqslant}$ are defined as follows:

$$\begin{aligned} A &::= \ \mathbf{bool} \mid X \mid A \to A \mid \forall(X{\leqslant}A)B \mid \mathbf{Top} \\ t &::= \ \mathit{true} \mid \mathit{false} \mid x \mid t\ t \mid \lambda x{:}A.t \mid \Lambda X{\leqslant}A.t \mid tA \mid (t?t{:}t) \\ M &::= A \mid t \\ \Gamma &::= \emptyset \mid \Gamma, x : A \mid \Gamma, X \leqslant A \end{aligned}$$

A *type* is either a type constant **bool**, which has two values $\mathit{true}$ and $\mathit{false}$, or a type variable $X$, or a function type $A \to B$, or a type of the form $\forall(X{\leqslant}A)B$, which is the type of a function that can be applied to a type $C$ smaller than or equal to $A$ and that returns a result of type $B[X{:=}C]$, or a type **Top** which is the supertype of all the types. An *atomic type* is either a type variable, or **bool** or **Top**.

An *expression* $M$ is either a type $A$ or a term $t$.

Terms are composed of those of simply-typed lambda calculus, of the constants $true$, $false$, whose type is **bool**, of the application of a term to a type $tA$, of the

abstraction over a bounded type variable, $\Lambda X{\leqslant}A.t$, of the if-expression $b?e_1{:}e_2$.

A *context* is either an empty context $\emptyset$, or a context followed by a typing declaration $x : A$, or a context followed by a subtyping declaration $X \leqslant A$. Thus, a context $\Gamma$ is a sequence of typing and subtyping declarations. The domain of $\Gamma$, denoted by $dom(\Gamma)$, is the set $\{x \mid x : A \in \Gamma\} \cup \{X \mid X \leqslant A \in \Gamma\}$. We write $X \notin \Gamma$(and $x \notin \Gamma$) for $X \notin dom(\Gamma)$ (and $x \notin dom(\Gamma)$). A context $\Gamma$ is *well-formed* if, for any $X \in \Gamma$, $\Gamma \equiv \Gamma_1, X \leqslant A, \Gamma_2$ implies that $X \notin \Gamma_1$, and for any $x \in \Gamma$, $\Gamma \equiv \Gamma_1, x : A, \Gamma_2$ implies that $x \notin \Gamma_1$. In this paper, we assume that contexts are always well-formed. Define $\Gamma(X) = A$ if $X \leqslant A \in \Gamma$.

Quantifying over all the types smaller than **Top** is equivalent to quantifying over all the types. The notation $\forall X.A$ is used as the abbreviation of $\forall X{\leqslant}\mathbf{Top}.A$.

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

$$(\lambda x{:}A.t_1)t_2 \rightarrow_{\beta_1} t_1[x := t_2]$$
$$(\Lambda X{\leqslant}A.t)B \rightarrow_{\beta_2} t[X := B]$$
$$(true?e_1{:}e_2) \rightarrow_{if} e_1$$
$$(false?e_1{:}e_2) \rightarrow_{if} e_2$$

The $\beta_1, \beta_2$ and $\beta_{if}$-reduction relations are the reflexive and transitive closures of $\beta_1, \beta_2$ and $\beta_{if}$ respectively. The $\beta_1, \beta_2$ and $\beta_{if}$-conversion relations are the reflexive, symmetric and transitive closures of $\beta_1, \beta_2$ and $\beta_{if}$ respectively. Two terms $e_1, e_2$ are $\beta_{if}$-convertible if the pair $(e_1, e_2)$ belongs to the $\beta_{if}$-conversion relation.

The subtyping rules are same as kernel $F_{\leqslant}$[13]:

**Subtyping rules:**

$$\frac{X \leqslant B \in \Gamma}{\Gamma \vdash X \leqslant B}\ Prim \qquad \frac{}{\Gamma \vdash A \leqslant A}\ Refl$$

$$\frac{\Gamma \vdash A \leqslant B \quad \Gamma \vdash B \leqslant C}{\Gamma \vdash A \leqslant C}\ Trans \qquad \frac{\Gamma \vdash C \leqslant A \quad \Gamma \vdash B \leqslant D}{\Gamma \vdash A{\rightarrow}B \leqslant C{\rightarrow}D}\ {\rightarrow}_{\leqslant}$$

$$\frac{}{\Gamma \vdash A \leqslant \mathbf{Top}}\ \mathbf{Top}_{\leqslant} \qquad \frac{\Gamma, X \leqslant A \vdash B \leqslant C}{\Gamma \vdash \forall(X{\leqslant}A)B \leqslant \forall(X{\leqslant}A)C}\ \forall_{\leqslant}$$

Note that in the full $F_{\leqslant}$, the rule $\forall_{\leqslant}$ is

$$\frac{\Gamma, X \leqslant A \vdash B \leqslant C}{\Gamma \vdash \forall(X{\leqslant}A)B \leqslant \forall(X{\leqslant}A)C}\ \forall_{\leqslant}$$

It is proved in Ref. [12] that type checking in the full $F_{\leqslant}$ is undecidable.

The weak typing rules are formed by adding to the type system of $F_{\leqslant}$ the typing rules for if-expression and for the constants $true, false$.

**Weak Typing System (WTS)**

$$\frac{}{\Gamma \vdash_W true : \mathbf{bool}}\ True \qquad \frac{}{\Gamma \vdash_W false : \mathbf{bool}}\ False$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash_W x : A}\ Var \qquad \frac{\Gamma \vdash_W b : \mathbf{bool} \quad \Gamma \vdash_W e_1 : C \quad \Gamma \vdash_W e_2 : C}{\Gamma \vdash_W (b?e_1{:}e_2) : C}\ If$$

$$\frac{\Gamma, x : B \vdash_W e : A}{\Gamma \vdash_W \lambda x{:}B.e : B{\rightarrow}A}\ Lam \qquad \frac{\Gamma \vdash_W e_1 : B{\rightarrow}A \quad \Gamma \vdash_W e_2 : B}{\Gamma \vdash_W e_1 e_2 : A}\ App$$

$$\frac{\Gamma, X \leqslant B \vdash_W e : A}{\Gamma \vdash_W \Lambda X \leqslant B.e : \forall(X \leqslant B)A}\ Lam2 \qquad \frac{\Gamma \vdash_W e : \forall(X \leqslant B)A \quad \Gamma \vdash C \leqslant B}{\Gamma \vdash_W eC : A[X:=C]}\ App2$$

$$\frac{\Gamma \vdash_W e : A \quad \Gamma \vdash A \leqslant B}{\Gamma \vdash_W e : B}\ Subsumption$$

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

$$\Gamma, x : A \vdash B \leqslant C \ \Rightarrow\ \Gamma \vdash B \leqslant C.$$

## 5 Subtyping Checking

The subtyping and typing rules presented in the previous section do not directly corresponds to a deterministic type checking algorithm because the set of rules does not satisfy the subformula property. In the subtyping system, the reflexivity rule and the transitivity rule are responsible for the loss of this property. In Ref. [6], Curien and Ghelli has studied this problem and created an algorithmic subtyping system for the full $F_{\leqslant}$. The following algorithmic subtyping system is adapted from theirs.

**Algorithmic Subtyping System**

$$\frac{X \leqslant B \in \Gamma \quad \Gamma \vdash B \leqslant C}{\Gamma \vdash X \leqslant C}\ \text{A-Trans} \qquad \frac{\Gamma \vdash C \leqslant A \quad \Gamma \vdash B \leqslant D}{\Gamma \vdash A \to B \leqslant C \to D}\ \text{A-}\to_{\leqslant}$$

$$\frac{A \text{ is an atomic type}}{\Gamma \vdash A \leqslant A}\ \text{A-Refl} \qquad \frac{}{\Gamma \vdash A \leqslant \mathbf{Top}}\ \text{A-Top}$$

$$\frac{\Gamma, X \leqslant A \vdash B \leqslant C}{\Gamma \vdash \forall(X \leqslant A)B \leqslant \forall(X \leqslant A)C}\ \text{A-}\forall_{\leqslant}$$

Note that this system does not contain the transitivity rule. A reflexivity rule over atomic types replaces the general reflexivity rule. Both the transitivity (Trans) rule and the general reflexivity (Refl) rule are admissible[6,13]. This set of rules can be straightforwardly turned into an algorithm. Therefore the subtyping checking problem for $F_{\leqslant}$ is decidable[13].

**Lemma 5.1 (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. The subtyping checking problem is decidable.

Since the algorithmic subtyping system and the original subtyping system are equivalent, the notation $\Gamma \vdash A \leqslant B$ will be used to denote a subtyping judgment in either system.

Observe that typing declarations are irrelevant in the derivation of subtyping.

**Lemma 5.2 (Strengthening)**

$$\Gamma_1, x : A, \Gamma_2 \vdash B \leqslant C \ \Rightarrow\ \Gamma_1, \Gamma_2 \vdash B \leqslant C.$$

We list three more properties. Their proofs can be found in the Appendix. First, the extension of a context will not change subtyping.

**Lemma 5.3 (Weakening for subtyping)**

$$\Gamma \vdash A \leqslant B \ \wedge \ x \notin \Gamma \ \Rightarrow \Gamma, x : C \vdash A \leqslant B;$$
$$\Gamma \vdash A \leqslant B \ \wedge \ X \notin \Gamma \Rightarrow \Gamma, X \leqslant C \vdash A \leqslant B.$$

The narrowing property says that subtyping is preserved under the restriction of the range of the type variable $X$.

**Lemma 5.4 (Narrowing of subtyping)**

$$\Gamma_1, X \leqslant B, \Gamma_2 \vdash A \leqslant A' \ \wedge \ \Gamma_1 \vdash C \leqslant B \ \Rightarrow \ \Gamma_1, X \leqslant C, \Gamma_2 \vdash A \leqslant A'.$$

Subtyping is also preserved if a type variable is substituted by a type of same bound.

**Lemma 5.5 (Subtyping substitution)**

$$\Gamma_1, X \leqslant B, \Gamma_2 \vdash A \leqslant A' \wedge \Gamma_1 \vdash C \leqslant B \ \Rightarrow \ \Gamma_1, \Gamma_2[X := C] \vdash A[X := C] \leqslant A'[X := C].$$

## 6 Strong Typing and Type Checking

In Ref. [7], the type checking problem for $\lambda^{if}_{\leqslant}$ is solved with a reformulation of the WTS. We adapt the method here. It requires two non-trivial modifications.

First, in $\lambda^{if}_{\leqslant}$, the reformulated typing rule for a term variable is:

$$\frac{x : B_1 \to \cdots \to B_n \to A' \in \Gamma \quad \Gamma \vdash A' \leqslant A \quad \Gamma \vdash e_i : B_i}{\Gamma \vdash x e_1..e_n : A} \ ApV$$

In simply typed $\lambda$-calculus, the type of a variable is always of the form $B_1 \to \cdots \to B_n \to A'$. In $F_{\leqslant}$ the general form of a type is more complicated. First, a type is typically a mixture of arrow types and bounded quantifications. Second, application of a typing rule might involve substitution of type variables in a type. For instance, the type of $x$ could be $\forall(X_1 \leqslant A_1)A_2 \to \forall(X_3 \leqslant A_3)A_4 \to C$. Let $\Gamma = C_1 \leqslant A_1, a_2 : A_2[X_1{:=}C_1], C_3 \leqslant A_3[X_1{:=}C_1], a_4 : A_4[X_1{:=}C_1, X_3{:=}C_3]$, then $\Gamma \vdash x C_1 a_2 C_3 a_4 : C[X_1{:=}C_1, X_3{:=}C_3]$.

To cope with this complex type structure, we split the $ApV$ rule into three rules: $ApV\Gamma$, $ApV\to$ and $ApV\forall$ (see the typing rules below).

The second problem is how to deal with terms like $(\Lambda X \leqslant B.e)CN_1..N_n$. These terms resemble terms of the form $(\lambda x{:}B.e)e'N_1..N_n$, for which we had a typing rule in $\lambda^{if}_{\leqslant}$[7]:

$$\frac{\Gamma, x : B \vdash eN_1..N_n : A \quad \Gamma \vdash e' : B \quad x \notin Fv(N_i)}{\Gamma \vdash (\lambda x{:}B.e)e'N_1..N_n : A} \ Ap\lambda$$

One might be tempted to create a similar rule for $\Lambda$ application:

$$\frac{\Gamma, X \leqslant B \vdash eN_1..N_n : A \quad \Gamma \vdash C \leqslant B \quad X \notin Fv(N_i)}{\Gamma \vdash (\Lambda X \leqslant B.e)CN_1..N_n : A}$$

Unfortunately, this does not work because $X$ is local in $e$. Consider, for example, $e = \lambda y{:}X.e'$. Since the type of $N_1$ is irrelevant to $X$, $eN_1$ is not typable. To get around of this problem, we use the rule:

$$\frac{\Gamma \vdash e[X{:=}C]N_1..N_n : A \quad \Gamma \vdash C \leqslant B}{\Gamma \vdash (\Lambda X{\leqslant}B.e)CN_1..N_n : A}\ Ap\Lambda$$

The set of reformulated typing rules is:

**Strong Typing System (STS)**

$$\frac{}{\Gamma \vdash true : \mathbf{bool}}\ True$$

$$\frac{}{\Gamma \vdash false : \mathbf{bool}}\ False$$

$$\frac{x : A' \in \Gamma \quad \Gamma \vdash A' \leqslant A}{\Gamma \vdash x : A}\ ApV\Gamma$$

$$\frac{\Gamma \vdash xN_1..N_n : B{\to}A' \quad \Gamma \vdash A' \leqslant A \quad \Gamma \vdash e' : B}{\Gamma \vdash xN_1..N_ne' : A}\ ApV{\to}$$

$$\frac{\Gamma \vdash xN_1..N_n : \forall(X{\leqslant}B)A' \quad \Gamma \vdash A'[X{:=}C] \leqslant A \quad \Gamma \vdash C \leqslant B}{\Gamma \vdash xN_1..N_nC : A}\ ApV\forall$$

$$\frac{\Gamma, x : B \vdash eN_1..N_n : A \quad \Gamma \vdash e' : B \quad x \notin Fv(N_i)}{\Gamma \vdash (\lambda x{:}B.e)e'N_1..N_n : A}\ Ap\lambda$$

$$\frac{\Gamma \vdash e[X{:=}C]N_1..N_n : A \quad \Gamma \vdash C \leqslant B}{\Gamma \vdash (\Lambda X{\leqslant}B.e)CN_1..N_n : A}\ Ap\Lambda$$

$$\frac{\Gamma, x : B \vdash e : A \quad \Gamma \vdash B' \leqslant B}{\Gamma \vdash \lambda x{:}B.e : B' \to A}\ A\lambda$$

$$\frac{\Gamma, X \leqslant B \vdash e : A}{\Gamma \vdash \Lambda X{\leqslant}B.e : \forall(X{\leqslant}B)A}\ A\Lambda$$

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

The first two rules ($True, False$) are typing rules for the two boolean constants. The rules ($ApV\Gamma$, $ApV{\to}$, $ApV\forall$) types terms of the form $xN_1..N_k$. The rules $Ap\lambda$ and $Ap\Lambda$ are for applications starting with $\lambda$-abstraction and $\Lambda$-abstraction respectively. $A\lambda$ and $A\Lambda$ are for $\lambda$-abstraction and $\Lambda$-abstraction. The last rule is the typing rule for if-expression.

The following example illustrates a type checking derivation involving both polymorphic function and if-expression, where the two branches of the if-expression are of different subtypes of the same type C.

**Example 6.1 (Apply a polymorphic function to an if-expression)** Assume $\Gamma$ contains $b : \mathbf{bool}, e_1 : A_1, e_2 : A_2, A_1 \leqslant C, A_2 \leqslant C$, the following is a type checking derivation in STS for an application of a polymorphic identity function to an if-expression:

$$\dfrac{\dfrac{\Gamma, x{:}C{\vdash}x{:}C \qquad \dfrac{\Gamma{\vdash}b{:}\mathbf{bool} \quad \dfrac{e_1{:}A_1{\in}\Gamma \quad \Gamma{\vdash}A_1{\leqslant}C}{\Gamma{\vdash}e_1{:}C}\ ApV\Gamma \quad \dfrac{\cdots}{\Gamma{\vdash}e_2{:}C}\ ApV\Gamma}{\Gamma{\vdash}b?e_1{:}e_2{:}C}\ ApIf}{\Gamma{\vdash}(\lambda x{:}C.x)(b?e_1{:}e_2){:}C}\ Ap\lambda \qquad \Gamma{\vdash}C{\leqslant}Top}{\Gamma{\vdash}(\Lambda X{\leqslant}Top.\lambda x{:}X.x)C(b?e_1{:}e_2){:}C}\ Ap\Lambda \tag{4}$$

The system STS is complete with respect to the weak type system:

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

This fact will be proved in Section 8.

On the other hand, the STS is strictly stronger than the WTS. This is mainly due to the $ApIf$ rule. The following is an example.

**Example 6.2 (A non-typable if-expression)** Assume a set of base types $X_1$, $X_2$, $Y_1, Y_2, Z$ and a set of predefined subtyping relation:

$$Y_1 \leqslant X_1,\ Y_2 \leqslant X_1,\ Y_1 \leqslant X_2,\ Y_2 \leqslant X_2$$

Note that $Y_1, Y_2$ have two common upper bounds $X_1, X_2$, but not any least common upper bound. Assume $b_1, b_2 : Bool, e_1 : X_1 \to Z, e_2 : X_2 \to Z, a_1 : Y_1, a_2 : Y_2$. Then the following typing is derivable in the weak type system:

$$(b_1?(e_1(b_2?a_1{:}a_2)) : (e_2(b_2?a_1{:}a_2))) : Z$$

The following term is $\beta_{if}$-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: $X_1$ and $X_2$, $(b_1?e_1{:}e_2)$ has two types $Y_1 \to Z$ and $Y_2 \to Z$, which are the only common upper bounds of $X_1 \to Z$ and $X_2 \to Z$. Neither $X_1$ nor $X_2$ is a subtype of $Y_1$ or $Y_2$.

Although the whole expression is typable, but the subterm $b_1?e_1{:}e_2$ is not typable, because $e_1$ and $e_2$ has different types, andy they do not have a common upper bound. Therefore, a typable expresion can have nontypable subexpressions.

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. Note that, it is a good thing to type as more terms as possible as long as these terms do not produce type errors at run time. The subject reduction theorem proved in this paper shows that STS is type safe.

## 7 Decidability of Type Checking

STS is close to an algorithmic type checking system. There are still two problems need to be solved. First, the rule $ApV \to$ and $ApV\forall$ do not have subformula properties: the type B in the assumptions of these rules does not appear in the conclusion. Therefore, STS does not directly correspond to a type checking algorithm. Second, assume the type checking problem for these two rules are solved, then we still need to prove the termination of the algorithm. The following two subsections solves these two problems.

### *7.1 Type checking algorithm*

As discussed above, the key to type checking is to check the judgement of form $\Gamma \vdash xN_1...N_k : A$ where $k > 0$. For this purpose, we extend the set of terms to include typed terms. Given a term $M$ and a type $A$, we introduce a *typed term* $\langle M : A \rangle$,

which means that the type $A$ is derived for the term $M$. The typed term is used in the type checking rules to record the derived types of subterms. The type checking problem is solved using the following rules:

$$\frac{x : B \in \Gamma \quad \Gamma \vdash \langle x : B \rangle N_1..N_k : A}{\Gamma \vdash xN_1..N_k : A} \qquad \frac{\Gamma \vdash A' \leqslant A}{\Gamma \vdash \langle M : A' \rangle : A}$$

$$\frac{\Gamma \vdash N_1 : B \quad \Gamma \vdash \langle tN_1 : D \rangle N_2..N_k : A}{\Gamma \vdash \langle t : B \to D \rangle N_1..N_k : A} \qquad \frac{\Gamma \vdash C \leqslant B \quad \Gamma \vdash \langle tC : D[X := C] \rangle N_3..N_k : A}{\Gamma \vdash \langle t : \forall X \leqslant B.D \rangle CN_2..N_k : A}$$

$$\frac{X \leqslant B \in \Gamma \quad \Gamma \vdash \langle t : B \rangle N_1..N_k : A}{\Gamma \vdash \langle t : X \rangle N_1..N_k : A}$$

### *7.2 Termination of type checking*

Most typing rules in STS has the property that the size of any term in the assumption is strictly smaller than that of the term in the conclusion. An exception is the rule $Ap\Lambda$, in which the size of the term $e[X{:=}C]N_1..N_n$ in the hypothesis might be larger than that of $(\Lambda X{\leqslant}B.e)CN_1..N_n$ in the conclusion. This shows that, the size alone is not enough to be used as a measure to prove the termination of type checking. Hence, we introduce a measure which combines the $\beta_2\Gamma$-reduction (defined below) with size of term.

Let $\Gamma$ be a context. The *one step* $\Gamma$*-reduction*, denoted by $\to_\Gamma$, is defined as the compatible closure of the following reduction:

$$X \to_\Gamma \Gamma(X)$$

The *one step* $\beta_2\Gamma$*-reduction*, denoted by $\to_{\beta_2\Gamma}$ is either one step $\beta_2$-reduction or one step $\Gamma$-reduction. The $\beta_2\Gamma$*-reduction* is the compatible closure of $\beta_2$- and $\Gamma$-reduction.

The $\beta_2$-reduction will not introduce new $\beta_2$-redex. Thus,

**Lemma 7.1 ($\beta_2$ Strong normalization)** The $\beta_2$-reduction is strongly normalizing.

$\Gamma$-reduction may introduce new $\Gamma$-redex. But a well-formed context $\Gamma$ does not have circularities, so $\Gamma$-reduction is obviously strongly normalizing.

**Lemma 7.2 ($\Gamma$ strong normalization)** If $\Gamma \vdash e : A$, then $\Gamma$-reduction from $e$ is terminating.

Since $\beta_2$-reduction may introduce $\Gamma$-redex and $\Gamma$-reduction may introduce a $\beta_2$-redex, so the proof of the termination of the combined $\beta_2\Gamma$-reduction is not trivial.

**Lemma 7.3 ($\beta_2\Gamma$ strong normalization)** If $\Gamma \vdash e : A$, then $\beta_2\Gamma$-reduction from $e$ is terminating.

*Proof.* Similar to the proof of $\beta_2\Gamma$-strong normalization in Ref. [5] although the $\beta_2$-reduction in that article is different from the $\beta_2$-reduction in this paper. □

Let $\Gamma$ be a context and $t$ a term. We denote by $MaxRed_\Gamma(e)$ the maximal length of a $\beta_2\Gamma$-reduction starting from $e$ and by $Size(e)$ the number of symbols appearing in $e$.

**Theorem 7.4 (Termination)** The type checking algorithm defined by the strong type system terminates.

*Proof.* First, the subtyping checking is decidable. Second, define a lexicographic order on terms: $Weight(e) = \langle MaxRed_\Gamma(e), Size(e)\rangle$. It is easy to verify that, in any typing rule of STS, any term in the premises is smaller than the term in the conclusion by this measure. Therefore, the type checking algorithm defined by STS terminates. □

Therefore, the type checking problem for $F^{if}_{\leqslant}$ is decidable.

## 8 Completeness of Strong Typing

In this section, we show that all typing rules are admissible in STS. Therefore, any term typable in WTS is also typable in STS.

**Lemma 8.1 (Admissibility of $Var$ and $Lam$)** The rule $Var$ and the rule $Lam$ in WTS are admissible in STS.

*Proof.* Use the fact that the reflexivity subtyping rule is admissible. □

**Lemma 8.2 (Admissibility of subsumption)**

$$\Gamma \vdash t : A \ \wedge\ \Gamma \vdash A \leqslant B \quad \Rightarrow \quad \Gamma \vdash t : B.$$

*Proof.* Induction on the size of $t$. $t$ can be written in the form of $e'N_1..N_n$ where $e'$ is not an application. The proof proceeds by analyzing the form of $e'$.

We treat the cases where $t = \lambda y{:}B'.t'$ and $t = \Lambda X{\leqslant}B'.t'$. Other cases are similar. Case $t = \lambda y{:}B'.t'$. Then $A, B$ must be arrow types: $A \equiv A_1 \to B_1, B \equiv A_2 \to B_2$, and

$$\begin{array}{lll}
\Gamma{\vdash}\lambda y : B'.t' : A_1{\to}B_1 \Rightarrow \Gamma, y : B'{\vdash}t' : B_1 \wedge \Gamma{\vdash}A_1{\leqslant}B' & A\lambda \\
\Gamma{\vdash}A_1{\to}B_1{\leqslant}A_2{\to}B_2 \Rightarrow \Gamma{\vdash}A_2{\leqslant}A_1 \wedge \Gamma{\vdash}B_1{\leqslant}B_2 & \\
\qquad\qquad \Rightarrow \Gamma{\vdash}A_2{\leqslant}B' \wedge \Gamma, y : B'{\vdash}B_1{\leqslant}B_2 & \textit{transitivity \& weakening} \\
\qquad\qquad \Rightarrow \Gamma, y : B'{\vdash}t' : B_2 & IH \\
\qquad\qquad \Rightarrow \Gamma{\vdash}\lambda y{:}B'.t' : A_2{\to}B_2 & A\lambda
\end{array}$$

Case $t = \Lambda X{\leqslant}B'.t'$ and $A = \forall(X{\leqslant}B')A'$. Then,

$$\begin{array}{ll}
\Gamma \vdash \Lambda X{\leqslant}B'.t' : \forall(X{\leqslant}B')A' & \textit{assumption} \\
\Rightarrow \Gamma, X \leqslant B' \vdash t' : A' & A\Lambda
\end{array}$$

$$\begin{array}{ll}
\Gamma \vdash \forall(X{\leqslant}B')A' \leqslant B & \textit{assumption} \\
\Rightarrow B \equiv \forall(X{\leqslant}B')A'' \ \wedge\ \Gamma, X \leqslant B' \vdash A' \leqslant A'' & \text{A-}\forall_{\leqslant} \\
\Rightarrow \Gamma, X \leqslant B' \vdash t' : A'' & IH \\
\Rightarrow \Gamma \vdash \Lambda X{\leqslant}B'.t' : \forall(X{\leqslant}B')A'' & A\Lambda \\
\Rightarrow \Gamma \vdash \Lambda X{\leqslant}B'.t' : B & B \equiv \forall(X{\leqslant}B')A''
\end{array}$$

□

**Lemma 8.3 (Admissibility of term application)**

$$\Gamma \vdash t : A \to B \ \wedge\ \Gamma \vdash e : A \quad \Rightarrow \quad \Gamma \vdash t\ e : B.$$

*Proof.* Induction on the size of $t$. $t$ can be written in the form of $e'N_1..N_n$ where $e'$ is not an application. The proof proceeds by analyzing the form of $e'$.

We treat the case where $t = (\Lambda X{\leqslant}B'.t')CN_1...N_n$, others are similar. Case $t = (\Lambda X{\leqslant}B'.t')CN_1..N_n$. Then

$$\begin{array}{ll} \Gamma \vdash (\Lambda X{\leqslant}B'.t')CN_1..N_n : A \to B & assumption \\ \Rightarrow \Gamma \vdash t'[X{:=}C]N_1..N_n : A \to B \ \wedge\ \Gamma \vdash C \leqslant B' & Ap\Lambda \\ \Rightarrow \Gamma \vdash t'[X{:=}C]N_1..N_ne : B & IH \\ \Rightarrow \Gamma \vdash (\Lambda X{\leqslant}B'.t')CN_1..N_ne : B & Ap\Lambda \end{array}$$

□

**Lemma 8.4 (Type substitution)**

$$\Gamma_1, X \leqslant B, \Gamma_2 \vdash t : A \ \wedge\ \Gamma_1 \vdash C \leqslant B \ \Rightarrow\ \Gamma_1, \Gamma_2[X := C] \vdash t[X := C] : A[X := C].$$

*Proof.* Let $\Gamma = \Gamma_1, X \leqslant B, \Gamma_2$. The proof proceeds by induction on the derivation of $\Gamma_1, X \leqslant B, \Gamma_2 \vdash t : A$. We give just the interesting cases.

Case $ApV\Gamma$. Assume that the last step of the derivation is

$$\frac{x : A' \in \Gamma \quad \Gamma \vdash A' \leqslant A}{\Gamma \vdash x : A}\ ApV\Gamma$$

By Lemma 5.5, $\Gamma_1, \Gamma_2[X := C] \vdash A'[X := C] \leqslant A[X := C]$.

$$\begin{array}{l} x : A' \in \Gamma \\ \Rightarrow x : A'[X := C] \in \Gamma_1, \Gamma_2[X := C] \\ \Rightarrow \Gamma_1, \Gamma_2[X := C] \vdash x : A[X := C]\ \ ApV\Gamma \end{array}$$

Case $ApV\forall$. Assume that the last step of the derivation is

$$\frac{\Gamma \vdash xN_1..N_n : \forall(Y{\leqslant}B')A' \quad \Gamma \vdash A'[X{:=}C'] \leqslant A \quad \Gamma \vdash C' \leqslant B'}{\Gamma \vdash xN_1..N_nC : A}\ ApV\forall$$

By induction assumption, we have

$$\begin{array}{lll} \Gamma \vdash xN_1..N_n : \forall(Y{\leqslant}B')A' & & \\ \Rightarrow \Gamma_1, \Gamma_2[X{:=}C] \vdash xN_1[X{:=}C]..N_n[X{:=}C] : \forall(Y{\leqslant}B'[X{:=}C])A'[X{:=}C] & IH & (1) \\ \Gamma \vdash A'[Y{:=}C'] \leqslant A & & \\ \Rightarrow \Gamma_1, \Gamma_2[X{:=}C] \vdash A'[Y{:=}C'][X{:=}C] \leqslant A[X{:=}C] & & \\ \Rightarrow \Gamma_1, \Gamma_2[X{:=}C] \vdash (A'[X{:=}C])[Y{:=}C'[X{:=}C]] \leqslant A[X{:=}C] & IH & (2) \\ \Gamma \vdash C' \leqslant B' & & \end{array}$$

$$\begin{array}{ll}
\Rightarrow \Gamma_1, \Gamma_2[X{:=}C] \vdash C'[X{:=}C] \leqslant B'[X{:=}C] & IH \quad (3) \\
\quad (1) + (2) + (3) & \\
\Rightarrow \Gamma_1, \Gamma_2[X{:=}C] \vdash xN_1[X{:=}C]..N_n[X{:=}C]C'[X{:=}C] : A[X{:=}C] & ApV\forall
\end{array}$$

□

The next lemma shows that the rule *App*2 in WTS is also admissible.

**Lemma 8.5 (Admissibility of type application)**

$$\Gamma \vdash t : \forall(X \leqslant B)A \ \wedge \ \Gamma \vdash C \leqslant B \ \Rightarrow \ \Gamma \vdash tC : A[X{:=}C].$$

*Proof.* Induction on the depth of the derivation of $\Gamma \vdash t : \forall(X \leqslant B)A$. $t$ can be written in the form of $t'N_1..N_n$ where $t'$ is not an application. The proof proceeds by analyzing the form of $t'$.

We treat the case where $t = \Lambda Y \leqslant B'.t$, others are similar.
Case $t = \Lambda Y \leqslant B'.t$.

$$\begin{array}{ll}
\quad \Gamma \vdash \Lambda X \leqslant B'.t' : \forall(X \leqslant B')A & assumption \\
\Rightarrow \Gamma, X \leqslant B' \vdash t' : A & A\Lambda \\
\Rightarrow \Gamma \vdash t'[X := C] : A[X := C] & Lemma\ 8.4 \\
\Rightarrow \Gamma \vdash (\Lambda X \leqslant B'.t')C : A[X := C] & Ap\Lambda
\end{array}$$

□

Since all typing rules in WTS are admissible in STS, so we have the completeness of the STS with respect to the WTS.

**Theorem 8.6 (Completeness of strong typing)** *Any typing judgment derivable in the WTS is also derivable in the STS.*

## 9 Subject Reduction

The substitution lemma is the key to the proof of subject reduction. The proof of this lemma uses the admissibility of subsumption and the admissibility of application.

**Lemma 9.1 (Term substitution)**

$$\Gamma_1, x : B, \Gamma_2 \vdash t : A \ \wedge \ \Gamma_1 \vdash e : B \ \Rightarrow \ \ \Gamma_1, \Gamma_2 \vdash t[x := e] : A.$$

*Proof.* Let $\Gamma = \Gamma_1, X \leqslant B, \Gamma_2$ and $\Gamma' = \Gamma_1, \Gamma_2$. Induction on the derivation depth of the judgement $\Gamma_1, x : B, \Gamma_2 \vdash t : A$. We treat only the cases where $t = x$ and $t = xN_1..N_ne'$.

Case $t = x$. Then,

$$\begin{array}{ll}
\quad \Gamma_1, x : B, \Gamma_2 \vdash x : A \ \wedge \Gamma_1 \vdash e : B & assumption \\
\Rightarrow \Gamma \vdash B \leqslant A \ \wedge \ \Gamma' \vdash e : B & ApV\Gamma\ \&\ weakening \\
\Rightarrow \Gamma' \vdash B \leqslant A & strengthening \\
\Rightarrow \Gamma' \vdash e : A & Lemma\ 8.2
\end{array}$$

Case $t = xN_1..N_ne'$. Then,

$$\begin{array}{ll} \Gamma \vdash xN_1..N_ne' : A & assumption \\ \Rightarrow \Gamma \vdash xN_1..N_n : B' \to A' \ \wedge\ \Gamma \vdash A' \leqslant A \ \wedge\ \Gamma \vdash e' : B' & ApV\to \\ \Rightarrow \Gamma' \vdash eN_1[x := e]..N_n[x := e] : B' \to A \ \wedge\ \Gamma' \vdash e'[x := e] : B' & IH \\ \Rightarrow \Gamma' \vdash eN_1[x := e]..N_n[x := e]e'[x := e] : A & Lemma\ 8.3 \\ \Rightarrow \Gamma' \vdash t[x := e] : A & \end{array}$$

□

The main result of this section is the subject reduction, which states that a well-typed term preserves its type during computation.

**Theorem 9.2 (Subject reduction)**

$$\Gamma \vdash t : A \ \wedge\ t \to_{\beta if} t' \ \Rightarrow\ \Gamma \vdash t' : A.$$

*Proof.* Induction on the derivation of $\Gamma \vdash t : A$. We treat the case where the last rule used in the derivation is $Ap\lambda$:

$$\frac{\Gamma, x : B \vdash eN_1..N_n : A \quad \Gamma \vdash e_1 : B \quad x \notin Fv(N_i)}{\Gamma \vdash (\lambda x{:}B.e)e_1N_1..N_n : A}\ Ap\lambda$$

There are three cases.

Case $e_1 \to_{\beta if} e_1'$. By induction hypothesis, $\Gamma \vdash e_1' : B$, the result follows.

Case $e \to_{\beta if} e'$ or $N_i \to_{\beta if} N_i' \quad i = 1..n$. Assume $e \to_{\beta if} e'$. By induction hypothesis, $\Gamma, x : B \vdash e'N_1..N_n : A$, the result follows. The cases for $N_i \to_{\beta if} N_i'$ $i = 2..n$ are similar.

Case $(\lambda x{:}B.e)e_1N_1..N_n \to_\beta e[x{:=}e_1]N_1..N_n$. Since $x \notin Fv(N_1, .., N_n)$, we have

$$\begin{array}{ll} \Gamma, x : B \vdash eN_1..N_n : A \ \wedge\ \Gamma \vdash e_1 : B & \\ \Rightarrow \Gamma \vdash e[x{:=}e_1]N_1..N_n : A & Lemma\ 9.1 \end{array}$$

□

## 10 Conclusions

**Summary** In this paper, we presented a calculus which combines $F_{\leqslant}$ and if-expression. The type checking problem for such a combination has been open for a long time. In our previous work, we have integrated if-expression with full subsumption in simply typed $\lambda$-calculus.This paper investigated the problem in System-F with bounded quantification. We formulated two type systems. The first one, named Weak Typing System (WTS), is constructed simply by adding a typing rule for if-expression to $F_{\leqslant}$. The second one, named Strong Typing System (STS), is formed in such a way that it is stronger than WTS and it corresponds directly to a type checking algorithm. It is shown that the type checking problem for STS is decidable and STS enjoys subject reduction.

There is a fundamental contribution in this paper. Traditionally, type checking relies on the derivation of a representative type, such as principle type or minimum

type. This is because the application rule does not have subformula property:

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \; \mathit{Apply}$$

In this rule, the type A in the assumption does not appear in the conclusion. Therefore, from the typing judgement in the conclusion, we need to find the type A in the assumption. That's why traditional type checking methods all rely on the derivation of a representative type from a given term.

The assumptions behind this approach are the existence of a unique representative type for a given term and the fact that all types of this term can be instanciated from its representative type. At the presence of both subtyping and if-expression, this assumption may not always be valid. This paper shows that it is possible to make a type checking algorithm not relying on the derivation of representative type. Thus, it opens a new direction for type checking. It is our hope that this method could be adapted to other situations where traditional type checking technique fails to work.

Studies on combining bounded quantification and subtyping were very active during 90's till the beginning of this century, but few works published since then.

## References


[1] Bruce KB, Crabtree J, Murtagh TP, van Gent R, Dimock A, Muller R. Safe and decidable type-checking in an object-oriented language. OOPSLA '93. 1993.

[2] Bruce KB, Longo G. A modest model of records, inheritance, and bounded quantification. Information and Computation, 1990, 87: 196–240.

[3] Breazu-Tannen V, Coquand T, Gunter C, Scedrov A. Inheritance as implicit coercion. Information and Computation, July 1991, 93(1): 172–221.

[4] Castagna G. Object-Oriented Programming: A Unified Foundation. Progress in Theoretical Computer Science. Birkäuser, Boston, 1997. ISBN 3–7643–3905–5.

[5] Castagna G, Chen G. Dependent types with subtyping and late-bound overloading. Information and Computation, 2001, 168: 1–67.

[6] Curien P-L, Ghelli G. Coherence of subsumption, minimum typing and the type checking in $F_{\leqslant}$. Mathematical Structures in Computer Science, 1992, 2(1): 55–91.

[7] Chen G. Full integration of subtyping and if-expression. The 4th Conference of Principle and Practice of Declarative Programming Languages(PPDP), October 2002. Pittsburgh, Pennsylvania, USA.

[8] Cardelli L, Longo G. A semantic basis for Quest. Journal of Functional Programming, 1991, 1(4): 417–458.

[9] Cardelli L, Martini S, Mitchell JC, Scedrov A. An extension of system F with subtyping. In Ito T, Meyer AR, eds. Theoretical Aspects of Computer Software. Springer-Verlag, September 1991. LNCS 526, 750–771.

[10] Cardelli L, Wegner P. On understanding types, data abstraction, and polymorphism. Computing Surveys, December 1985, 17(4): 471–522.

[11] Hosoya H, Pierce B, Turner D. subject reduction fails in java, June 1998. Message on Types Electronic Mailing List.

[12] Pierce BC. Bounded quantification is undecidable. Information and Computation, 1993, 112(1): 131–165.

[13] Pierce BC. Types and Programming Languages. MIT Press, 2002.

## Appendix A Structural Properties

This appendix contains the proofs of the lemmas mentioned in Section 5 and the one required in Section 8.

**Lemma A.1 (Narrowing)** *Let* $\Gamma \equiv \Gamma_1, x : C, \Gamma_2$ *and* $\Gamma' \equiv \Gamma_1, x : C', \Gamma_2$.

$$\Gamma \vdash t : A \ \wedge \ \Gamma_1 \vdash C' \leqslant C \ \Rightarrow \ \Gamma' \vdash t : A$$

*Proof.* Induction on the derivation of the judgement $\Gamma \vdash t : A$. We analyze two cases $ApV\Gamma$ and $ApV{\to}$. Others are similar.

Case $ApV\Gamma$. Assume that $t = y$ and the last step of the derivation of $\Gamma \vdash y : A$ is

$$\frac{y : A' \in \Gamma \quad \Gamma \vdash A' \leqslant A}{\Gamma \vdash y : A} \ ApV\Gamma$$

There are two subcases.

Subcase $y \neq x$. Then $y : A' \in \Gamma_1, \Gamma_2$, so $y : A' \in \Gamma'$. Since subtyping is irrelevant to typing, we have $\Gamma' \vdash A' \leqslant A$. It follows from $ApV\Gamma$ that $\Gamma' \vdash y : A$.

Subcase $y = x$. Therefore $C = A'$. By assumption $\Gamma' \vdash C' \leqslant A'$. Using transitivity and the fact that $\Gamma' \vdash A' \leqslant A$, we get $\Gamma' \vdash C' \leqslant A$. Hence we have the derivation:

$$\frac{y : C' \in \Gamma' \quad \Gamma' \vdash C' \leqslant A}{\Gamma' \vdash y : A} \ ApV\Gamma$$

Case $ApV{\to}$. Assume that $t = yN_1..N_ne'$ and that the last step of the derivation for $\Gamma \vdash yN_1..N_ne' : A$ is

$$\frac{\Gamma \vdash yN_1..N_n : B{\to}A' \quad \Gamma \vdash A' \leqslant A \quad \Gamma \vdash e' : B}{\Gamma \vdash yN_1..N_ne' : A} \ ApV{\to}$$

By induction assumption, $\Gamma' \vdash yN_1..N_n : B{\to}A'$ and $\Gamma' \vdash e' : B$. On the other hand, the typing declaration is irrelevant to subtyping, so $\Gamma' \vdash A' \leqslant A$. It follows from $ApV{\to}$ that $\Gamma' \vdash yN_1..N_ne' : A$. □

**Lemma A.2 (Weakening)**

1. $\Gamma \vdash t : B \ \wedge \ x \notin \Gamma \ \Rightarrow \ \Gamma, x : A \vdash t : B$;
2. $\Gamma \vdash t : B \ \wedge \ X \notin \Gamma \ \Rightarrow \ \Gamma, x \leqslant A \vdash t : B$.

*Proof.* Induction on the typing derivations. □

**Lemma A.3 (Generation of subtyping)**

1. $\Gamma \vdash A \to B \leqslant C \to D \ \Rightarrow \ \Gamma \vdash C \leqslant A \ \wedge \ \Gamma \vdash B \leqslant D$;
2. $\Gamma \vdash \forall(X{\leqslant}A).B \leqslant \forall(X{\leqslant}A).C \ \Rightarrow \ \Gamma, X \leqslant A \vdash B \leqslant C$.

*Proof.* 1. Just note that, in the algorithmic version of the subtyping system, the only way to derive $\Gamma \vdash A \to B \leqslant C \to D$ is through the rule A-$\to_{\leqslant}$; 2. Similar. □

**Lemma A.4 (Generation of typing)**

1. $\Gamma \vdash x : A \ \Rightarrow \ \ x : B \in \Gamma \ \wedge \ \Gamma \vdash B \leqslant A$;
2. $\Gamma \vdash \lambda x{:}B.e : A \ \Rightarrow \ \ A \equiv C \to D \ \wedge \ \Gamma \vdash C \leqslant B \ \wedge \ \Gamma, x : B \vdash e : D$;
3. $\Gamma \vdash e\ t : A \ \wedge e \not\equiv (b?e_1{:}e_2)N_1..N_n \ \Rightarrow \ \ \Gamma \vdash e : B \to A \ \wedge \ \Gamma \vdash t : B$;
4. $\Gamma \vdash \Lambda X{\leqslant}B.e : A \ \Rightarrow \ \ A \equiv \forall(X \leqslant B).C \ \wedge \ \Gamma, X \leqslant B \vdash e : C$;
5. $\Gamma \vdash e\ C : A \ \wedge e \not\equiv (b?e_1{:}e_2)N_1..N_n$
   $\Rightarrow \ \exists B, D.\ s.t.\ \Gamma \vdash C \leqslant B \ \wedge \ \Gamma \vdash e : \forall(X{\leqslant}B).D \ \wedge \ A = D[X{:=}C]$;
6. $\Gamma \vdash (b?e_1{:}e_2)N_1..N_n : A \ \ \Rightarrow \ \Gamma \vdash e_1N_1..N_n : A \wedge \Gamma \vdash e_2N_1..N_n : A \wedge \Gamma \vdash b : \mathit{Bool}$.

*where* $n \geqslant 0$.

*Proof.* We give the proof of the property 3., others are similar.

The assertion can be proved by induction on the derivation of $\Gamma \vdash e\ t : A$. Note that this typing judgement can only be derived via rules $\mathit{ApV}{\to}$, $\mathit{Ap}\lambda$ and $\mathit{Ap}\Lambda$.
Case $\mathit{ApV}{\to}$. Let $e = xN_1..N_n$ and $t = e'$. Assume that the last step in the derivation of $\Gamma \vdash e\ t : A$ is

$$\frac{\Gamma \vdash xN_1..N_n : B{\to}A' \quad \Gamma \vdash A' \leqslant A \quad \Gamma \vdash e' : B}{\Gamma \vdash xN_1..N_ne' : A}\ \mathit{ApV}{\to}$$

From $\Gamma \vdash A' \leqslant A$, it follows that $\Gamma \vdash B \to A' \leqslant B \to A$. By induction assumption and the fact that $\Gamma \vdash xN_1..N_n : B{\to}A'$, we get $\Gamma \vdash xN_1..N_n : B{\to}A$, which is the desired result.
Case $\mathit{Ap}\lambda$. Let $e = (\lambda x{:}B.e)e'N_1..N_{n-1}$ and $t = N_n$. Assume that the last step in the derivation of $\Gamma \vdash e\ t : A$ is:

$$\frac{\Gamma, x : B \vdash eN_1..N_n : A \quad \Gamma \vdash e' : B \quad x \notin Fv(N_i, e')}{\Gamma \vdash (\lambda x{:}B.e)e'N_1..N_n : A}\ \mathit{Ap}\lambda$$

By induction assumption, there is a type $C$ such that $\Gamma, x : B \vdash eN_1..N_{n-1} : C \to A$ and $\Gamma, x : B \vdash N_n : C$. Apply the rule $\mathit{Ap}\lambda$, we get $\Gamma \vdash (\lambda x{:}B.e)e'N_1..N_{n-1} : C \to A$. The assertion is proved.

Case $\mathit{Ap}\Lambda$. Let $e = (\Lambda X{\leqslant}B.e'')CN_1..N_{n-1}$ and $t = N_n$. Assume that the last step in the derivation of $\Gamma \vdash e\ t : A$ is:

$$\frac{\Gamma \vdash e''[X{:=}C]N_1..N_n : A \quad \Gamma \vdash C \leqslant B}{\Gamma \vdash (\Lambda X{\leqslant}B.e'')CN_1..N_n : A}\ \mathit{Ap}\Lambda$$

By induction assumption, there is a type $C'$ such that $\Gamma \vdash e''[X{:=}C]N_1..N_{n-1} : C' \to A$ and $\Gamma \vdash N_n : C'$. Apply the rule $\mathit{Ap}\Lambda$, we get $\Gamma \vdash (\Lambda X{\leqslant}B.e'')CN_1..N_{n-1} : C' \to A$. The assertion is proved. □

Note that the condition $e \not\equiv (b?e_1{:}e_2)N_1..N_n$ is necessary in the third property. The assertion

$$\Gamma \vdash e\ t : A \Rightarrow \exists B.\ s.t.\ \Gamma \vdash e : B \to A \ \wedge \ \Gamma \vdash t : B$$

is not valid in general. A counterexample is the term $(b_1?e_1{:}e_2)(b_2?a_1{:}a_2)$ in Example 6.2. Similar argument applies to the fifth property.