System F^ω with subtyping

Jakub Tarnawski and Damian Straszak

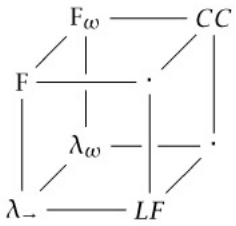
Wrocław, February 16, 2011

Contents

1 Introduction

The system F^ω (Girard, 1972) is a typed λ-calculus with higher-order polymorphism. Unlike $\lambda_\rightarrow$ and system F, it includes facilities for abstraction and application within type expressions, thus introducing type operators. To guarantee the well-formedness of type expressions built using the new machinery, an extra level of **kinds** is added. The kind $*$ classifies ordinary ("proper") types that we know from simply typed λ-calculus. Kinds of the form $K_1 \Rightarrow K_2$ classify type operators: type-level functions which map types of kind K_1 to types of kind K_2.

F^ω is a member of the broader family of calculi called the λ-cube. It already contains polymorphism and type operators, and augmenting it with dependent types would yield the calculus of constructions. For more information about the lambda-cube and its generalization - the pure type systems, see [4] and [5].



Instead, we extend F^ω with subtyping (yielding the system denoted $F^\omega_{<:}$). To that end, we introduce a pre-ordering $S <: T$ on elements of each kind K. Now, the appearance of each type variable X in a context Γ is augmented with an upper bound, written $X <: T$, which constrains X to range over the subtypes of T in the appropiate kind. We assume that the subtype relation has a maximal element on each kind K and denote it $Top[K]$. This ensures that well-typed terms in F^ω are still well-typed in our new system.

Let us start with a formal definition of $F^\omega_{<:}$.

2 Definitions

2.1 Syntax

Syntactic forms of the language under consideration can be summarized compactly by the following grammar:

$T ::=$	*types*
Top	*maximum type*
X	*type variable*
$T \to T$	*type of functions*
$\forall X <: T.T$	*universal type*
$\lambda X :: K.T$	*operator abstraction*
$T\ T$	*type application*
$Bool$	*type of Booleans*
Nat	*type of natural numbers*
$\{l_i = T_i\}_{i=1}^{n}$	*type of records*

$t ::=$	*terms*
x	*variable*
$\lambda x.t$	*abstraction*
$t\ t$	*application*
$\lambda X <: T.t$	*type abstraction*
$t\ [T]$	*type application*
$unit$	*constant unit*
$true$	*constant true*
$false$	*constant false*
$nat\ n$	*natural number*
$if\ t\ then\ t\ else\ t$	*conditional*
$\{l_i = t_i\}_{i=1}^{n}$	*record*
$t.l$	*projection*
$let\ x = t\ in\ t$	*let binding*
$let\ rec\ x = t\ in\ t$	*let binding*
$fix\ t$	*fixed point operator*
$succ\ t$	*successor*
$pred\ t$	*predecessor*
$letop\ X = T\ in\ t$	*let type-operator binding*

$K ::=$	*kinds*
$*$	*kind of proper types*
$K \Rightarrow K$	*kind of operators*

$v ::=$	*values*
$unit$	*unit value*
$true$	*Boolean true value*
$false$	*Boolean false value*
$nat\ n$	*numeric value*
$\{l_i = v\}_{i=1}^{n}$	*record value*
$\lambda x.t$	*abstraction value*
$\lambda X <: T.t$	*type abstraction value*

This is essentially an extended version of the language considered in chapter 31.1 of [1]. We added Boolean and numeric expressions, conditionals, fixed point recursion, let bindings and records. Also, to make our typechecker implementation easier to use, we introduced the *letop* binding, which is merely syntactic sugar; it replaces all occurences of the given type variable with the given type body. The substitution is carried out during parsing.

2.2 Evaluation

For the sake of brevity we present only the inference rules which involve type abstraction and type application. Other rules are the same as in simply typed lambda-calculus and they can be found in chapters 9 and 11 of [1].

$$\frac{t_1 \to t_1'}{t_1\ [T_2] \to t_1'\ [T_2]}$$

$$\lambda X : T_1.t\ [T_2] \to [X \to T_2]\, t$$

2.3 Kinding

The kinding relation on types closely resembles the typing relation on terms in $\lambda_{\to}$.

$$\Gamma \vdash Top :: *$$

$$\Gamma \vdash Bool :: *$$

$$\Gamma \vdash Nat :: *$$

$$\frac{\text{for every i,} \quad \Gamma \vdash T_i :: *}{\Gamma \vdash \{l_i = T_i\}_{i=1}^{n} :: *}$$

$$\frac{X <: T \in \Gamma \qquad \Gamma \vdash T :: K}{\Gamma \vdash X :: K}$$

$$\frac{\Gamma, X <: Top[K_1] \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1.T_2 :: K_1 \Rightarrow K_2}$$

$$\frac{\Gamma \vdash T_1 :: K_1 \Rightarrow K_2 \qquad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1\ T_2 :: K_2}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *}$$

$$\frac{\Gamma, X <: T_1 \vdash T_2 :: *}{\Gamma \vdash \forall X <: T_1.T_2 :: *}$$

2.4 Type equivalence

$$T \equiv T$$

$$\frac{T \equiv S}{S \equiv T}$$

$$\frac{T \equiv S \qquad S \equiv U}{T \equiv U}$$

$$\frac{T_1 \equiv S_1 \qquad T_2 \equiv S_2}{T_1 \to S_1 \equiv T_2 \to S_2}$$

$$\frac{T_1 \equiv S_1 \qquad T_2 \equiv S_2}{\forall X <: T_1.T_2 \equiv \forall X <: S_1.S_2}$$

$$\frac{T \equiv S}{\lambda X :: K.T \equiv \lambda X :: K.S}$$

$$\frac{T_1 \equiv S_1 \qquad T_2 \equiv S_2}{T_1\ S_1 \equiv T_2\ S_2}$$

$$(\lambda X :: K.T_1)T_2 \equiv [X \to T_2]T_1$$

2.5 Subtyping

In the presentation of typing and subtyping, we skip rules for base types and records, which are standard and can be found in chapters 15-17 of [1].

$$\frac{\Gamma \vdash T <: S \qquad \Gamma \vdash S <: U \qquad \Gamma \vdash S :: K}{\Gamma \vdash T <: U}$$

$$\frac{\Gamma \vdash S :: K}{\Gamma \vdash S <: Top}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2}$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T}$$

$$\frac{\Gamma \vdash U_1 :: K \qquad \Gamma \vdash U_2 :: K \qquad \Gamma, X <: U_1 \vdash S <: T \qquad \Gamma \vdash U_1 <: U_2 \qquad \Gamma \vdash U_2 <: U_1}{\Gamma \vdash \forall X <: U_1.S <: \forall X <: U_2.T}$$

$$\frac{\Gamma, X <: Top[K] \vdash S <: T}{\Gamma \vdash \lambda X :: K.S <: \lambda X :: K.T}$$

$$\frac{\Gamma \vdash S <: T}{\Gamma \vdash S\ U <: T\ U}$$

$$\frac{\Gamma \vdash S :: K \qquad \Gamma \vdash T :: K \qquad S \equiv T}{\Gamma \vdash S <: T}$$

The fifth rule (the one for universal quantifiers) is the most interesting. For system $F_{<:}$, many versions of this rule have been proposed. There are two well-known flavours different than ours:

$$\frac{\Gamma \vdash U :: K \qquad \Gamma, X <: U \vdash S <: T}{\Gamma \vdash \forall X <: U.S <: \forall X <: U.T}$$

$$\frac{\Gamma \vdash U_1 :: K \qquad \Gamma \vdash U_2 :: K \qquad \Gamma, X <: U_1 \vdash S <: T \qquad \Gamma \vdash U_2 <: U_1}{\Gamma \vdash \forall X <: U_1.S <: \forall X <: U_2.T}$$

The flavour of the system that uses the first rule is called the *kernel version*, the second one - the *full version*. The intuition behind the full version is that a term whose type is universal can be thought of as a function from types to terms. As such, it should be only contravariant with respect to its type parameter. Of the three possibilities, the kernel version is the most restrictive, while the full version is most permissive. We use a version that is in-between (and call it kernel+) - it provides a slight edge over the kernel version in a language where subtyping equivalence classes are non-trivial (in our language, this is caused by the presence of records), but is still decidable, unlike the full version. For a more detailed discussion, see [1] (chapters 26.2, 28.4 and 28.5; particularly exercise 28.4.3, which contains the kernel+ version).

2.6 Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T \to S \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ t_2 : S}$$

$$\frac{\Gamma, X <: T \vdash t : S}{\Gamma \vdash \lambda X <: T.t : \forall X <: T.S}$$

$$\frac{\Gamma \vdash t : \forall X <: T.S \qquad \Gamma \vdash U <: T}{\Gamma \vdash t_1\ [U] : [X \to U]S}$$

$$\frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T \qquad \Gamma \vdash T :: *}{\Gamma \vdash t : T}$$

3 Metatheory and implementation

In this section, we mainly provide syntax-directed or algorithmic versions of the relations defined above. The proofs of their correctness or termination are either routine or very involved; we give references to literature for the longer ones.

3.1 Progress and preservation

Firstly, we are interested in the usual progress and preservation properties of our system. Let us state these theorems:

Theorem 1 (Progress)**.** *Suppose* $\vdash t : T$*. Then either* t *is a value or there exists* t' *with* $t \to t'$*.*

Theorem 2 (Preservation)**.** *Suppose* $\vdash t : T$ *and* $t \to t'$*. Then* $\vdash t' : T$*.*

Proof. The proofs (version for F^ω) can be found in [1], and similar arguments for $F^\omega_{<:}$ are in [2]. □

3.2 Kinding

Lemma 3 (Decidability of kinding)**.** *The relation* $\Gamma \vdash T :: K$ *is decidable.*

Lemma 4 (Uniqueness of kinding)**.** *If* $\Gamma \vdash T :: K_1$ *and* $\Gamma \vdash T :: K_2$ *then* $K_1 = K_2$*.*

Proof. From the kinding rules we can easily construct an algorithm which computes the kind of a given type or answers no if the type is not well-kinded. A straightforward inductive argument gives the correctness. The details are presented in [2]. □

3.3 Algorithmic subtyping

The subtyping relation introduced in section 1 directly expresses its intended meaning. It is clear that such a relation should behave like an ordering (actually, in a language with records, it is a pre-order); in particular, it should be transitive. This property is expressed by one of the rules. Unfortunately, in this form the relation is not directly implementable. It is not even obvious whether it is computable at all. To show that it is, and how, we first need some background.

Definition 5. First, we define $\beta\top$-reduction as the reduction on types. It is analogous to the usual evalution relation in $\lambda_\to$; its rigorous definition is given in chapter 2.3.1 of [2].

Definition 6. We will denote by $T^!$ the $\beta\top$-normalized form of type T.

Definition 7. Let us also define the promotion relation $\uparrow$ for a type $X\ S_1...S_n$ as $\Gamma(X)\ S_1...S_n$.

Now we present a subtyping algorithm from [2] (page 258) which is used in our implementation.

$$
\begin{array}{l}
check(\Gamma \vdash S <: T) = \\
\quad check^!(\Gamma \vdash S^! <: T^!)
\end{array}
$$

$$
\begin{array}{l}
check^!(\Gamma \vdash S <: T) = \\
\quad if\ T \equiv Top(Kind_\Gamma(S)) \\
\quad\quad then\ true \\
\quad else\ if\ S \equiv T \\
\quad\quad then\ true \\
\quad else\ if\ S \uparrow_\Gamma U \\
\quad\quad then\ check^!(\Gamma \vdash U^! <: T) \\
\quad else\ if\ S \equiv S_1 \to S_2\ and\ T \equiv T_1 \to T_2 \\
\quad\quad then\ check^!(\Gamma \vdash T_1 <: S_1)\ and\ check^!(\Gamma \vdash S_2 <: T_2) \\
\quad else\ if\ S \equiv \forall X <: U.S_2\ and\ T \equiv \forall X <: U.T_2 \\
\quad\quad then\ check^!(\Gamma, X <: U \vdash S_2 <: T_2) \\
\quad else\ if\ S \equiv \lambda X :: K.S_2\ and\ T \equiv \forall X :: K.T_2 \\
\quad\quad then\ check^!(\Gamma, X <: Top[K] \vdash S_2 <: T_2) \\
\quad else\ false
\end{array}
$$

The algorithm is correct, under the assumption that types S and T are well-kinded. This statement is supported by the following theorems:

Theorem 8. *Suppose $\Gamma \vdash S :: K_S$ and $\Gamma \vdash T :: K_T$. Then check halts when presented with $\Gamma \vdash S <: T$ as input. Proposition 6.6.1 from [2].)*

Proof. The proof is very involved, and so we will only present its main ideas. First, $\beta\top\Gamma$-reduction is defined (see section 5 of [2]), where, beside the usual β and $\top$ reductions, variables are allowed to be replaced by their upper type-bounds from the context. This is reminiscent of replacing a type definition by its expansion. The reduction is then shown to be strongly normalizing whenever the type to be reduced is well-kinded. We rank well-kinded types V in a context Γ by pairs built from the maximum length of a $\beta\top\Gamma$-reduction sequence starting from V and the number of characters in V, ordered lexicographically. Then we put the rank of a statement $\Gamma \vdash S <: T$ to be the sum of ranks for S and T. Then recursive calls in *check* reduce the rank of their input, with the exception of the third clause (the one for promotion). But we are only in trouble when it promotes its first argument to $Top[K]$ for some kind K. But in that case the algorithm will halt on the next step. □

Theorem 9. *The algorithm **check** is sound and complete with respect to the original subtyping relation (on well-kinded types). (Theorem 6.6.3 from [2]. See there for proof.)*

Remark 10. Another flavour of algorithmic subtyping for $F^{\omega}_{<:}$ is presented in the paper [3], where also much shorter proofs of correctness and termination can be found.

3.4 Typing algorithm

In this section we present the typing algorithm which is used in our implementation.

First let us introduce the notion of a minimal type. A type S is minimal for a term s in a context Γ if $\Gamma \vdash s : S$ and for each T such that $\Gamma \vdash s : T$ we have $\Gamma \vdash S <: T$.

We will also need the promote-normal form of T in Γ:

$$\uparrow^{!}_{\Gamma} T = \begin{cases} \uparrow^{!}_{\Gamma} U & \text{if } T^{!} \uparrow_{\Gamma} U \\ T^{!} & \text{if } T^{!} \text{ cannot be promoted} \end{cases}$$

We define the typing algorithm by the following inference rules (to distinguish the typing algorithm from the original typing relation we use the symbol $\vdash_A$):

$$\frac{x : T \in \Gamma}{\Gamma \vdash_A x : T}$$

$$\frac{\Gamma, x : T_1 \vdash_A e : T_2}{\Gamma \vdash_A \lambda x : T_1 . e : T_1 \to T_2}$$

$$\frac{\uparrow^{!}_{\Gamma} S = T_1 \to T_2 \qquad \Gamma \vdash_A s : S \qquad \Gamma \vdash_A t : T \qquad \Gamma \vdash T <: T_1}{\Gamma \vdash_A s\ t : T_2}$$

$$\frac{\Gamma, X <: T_1 \vdash_A e : T_2}{\Gamma \vdash_A \lambda X <: T_1 . e : \forall X <: T_1 . T_2}$$

$$\frac{\uparrow^{!}_{\Gamma} T = \forall X <: T_1 . T_2 \qquad \Gamma \vdash_A t : T \qquad \Gamma \vdash S :: K \qquad \Gamma \vdash S <: T_1}{\Gamma \vdash_A t\ [S] : [X \to S] T_2}$$

Theorem 11. *If* $\Gamma \vdash_A t : T$ *then* $\Gamma \vdash T :: *$ *and* $\Gamma \vdash t : T$.

If $\Gamma \vdash s : T$ *then* $\Gamma \vdash T :: *$ *and* $\Gamma \vdash_A s : S$ *where* S *is minimal for* s *in* Γ.

Proof. Found in [2]. □

3.5 Type equivalence

It is worth noting that the type equivalence definition also contains the transitivity rule. But in this case it is not so problematic. We can simply skip it (as well as the symmetry rule) without changing the induced relation. After this step we can easily make the relation algorithmic. We just apply the same trick as in previous sections: we always begin the computations with simplifying the types to their normal form. In [2], Pierce proves that such a reduction always terminates.

3.6 Typing conditionals

When implementing a typing relation in a system with subtyping, one of the interesting cases is the conditional. Intuitively, we want the type of $cond := if\ b\ then\ t\ else\ s$ to be as general as possible. Of course, during the typing phase we cannot check whether b will evaluate to $True$ or $False$, so we have to ensure that the type of *cond* is a supertype of types of both branches. It is thus reasonable to define the type of *cond* as their smallest common supertype (also called their join and written $T \vee S$). So the rule for the conditional is:

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash s : S \qquad \Gamma \vdash b : Bool \qquad U = T \vee S}{\Gamma \vdash if\ b\ then\ t\ else\ s : U}$$

We are left with the task of calculating $T \vee S$. Because the arrow operator is contravariant with respect to the parameter, we also need to be able to calculate meets (the biggest common supertype of two types) whenever they exist. Calculating meets requires, in turn, calculating joins, so we have to make our definitions mutually recursive. The join and meet calculation algorithms for the kernel version of $F_{<:}$ are presented in chapter 28.6 of [1], along with proofs of correctness. To modify those to work for our system $F^{\omega}_{<:}$, we need to account for the existence of type abstractions and applications. This is easily accomplished by simplifying both types at the beginning of each recursive call. We also provide the definition for joins and meets of record types, and replace the rule for the quantifier with the slightly more permissive kernel+ rule, as we did with subtyping.

For completeness, we present the resulting algorithm, along with statements of correctness. Their proofs remain the same as in [1], because S and T are always simplified (reduced to head normal form).

$$\Gamma \vdash S \vee T = \begin{cases} T & \text{if } \Gamma \vdash S <: T \\ S & \text{if } \Gamma \vdash T <: S \\ S^! \ \overline{\nabla}\ T^! & \text{otherwise} \end{cases}$$

$$\Gamma \vdash S \ \overline{\nabla}\ T = \begin{cases} J & \text{if } S = X, \quad X <: U \in \Gamma, \quad \Gamma \vdash U \vee T = J \\ J & \text{if } T = X, \quad X <: U \in \Gamma, \quad \Gamma \vdash S \vee U = J \\ M_1 \to J_2 & \text{if } S = S_1 \to S_2, \quad T = T_1 \to T_2, \\ & \quad \Gamma \vdash S_1 \wedge T_1 = M_1, \quad \Gamma \vdash S_2 \vee T_2 = J_2 \\ \forall X <: U_1 . J_2 & \text{if } S = \forall X <: S_1 . S_2, \quad T = \forall X <: T_1 . T_2, \\ & \quad \Gamma \vdash S_1 <: T_1, \quad \Gamma \vdash T_1 <: S_1, \quad \Gamma, X <: S_1 \vdash S_2 \vee T_2 = J_2 \\ \{l = U_l\}_{l \in I} & \text{if } S = \{l = S_l\}_{l \in I_S}, \quad T = \{l = T_l\}_{l \in I_T}, \quad I = I_S \cap I_T, \\ & \quad \text{for every } l \in I, \quad \Gamma \vdash S_i \vee T_i = U_i \\ \text{Top} & \text{otherwise} \end{cases}$$

$$\Gamma \vdash S \wedge T = \begin{cases} S & \text{if } \Gamma \vdash S <: T \\ T & \text{if } \Gamma \vdash T <: S \\ S^! \ \overline{\wedge}\ T^! & \text{otherwise} \end{cases}$$

$$\Gamma \vdash S \overline{\wedge} T = \begin{cases} J_1 \to M_2 & \text{if } S = S_1 \to S_2, \quad T = T_1 \to T_2, \\ & \quad \Gamma \vdash S_1 \vee T_1 = J_1, \quad \Gamma \vdash S_2 \wedge T_2 = M_2 \\ \forall X <: U_1 . M_2 & \text{if } S = \forall X <: S_1 . S_2, \quad T = \forall X <: T_1 . T2, \\ & \quad \Gamma \vdash S_1 <: T_1, \quad \Gamma \vdash T_1 <: S_1, \quad \Gamma, X <: S_1 \vdash S_2 \wedge T_2 = J_2 \\ \{l = U_l\}_{i \in J} & \text{if } S = \{l = S_l\}_{l \in I_S}, \quad T = \{l = T_l\}_{l \in I_T}, \quad I = I_S \cap I_T, \\ & \quad \text{for every } l \in I, \quad \Gamma \vdash S_i \wedge T_i = U_i, \\ & \quad J = I_S \cup I_T, \quad \text{for every } l \in J \setminus I, \quad U_i = S_i \text{ or } U_i = T_i \\ \text{fail} & \text{otherwise} \end{cases}$$

Theorem 12. *1. If $\Gamma \vdash S \vee T = J$, then $\Gamma \vdash S <: J$ and $\Gamma \vdash T <: J$.*

2. If $\Gamma \vdash S \wedge T = M$, then $\Gamma \vdash M <: S$ and $\Gamma \vdash M <: T$.

3. If $\Gamma \vdash S <: V$ and $\Gamma \vdash T <: V$, then $\Gamma \vdash S \vee T = J$ for some J with $\Gamma \vdash J <: V$.

4. If $\Gamma \vdash L <: S$ and $\Gamma \vdash L <: T$, then $\Gamma \vdash S \wedge T = M$ for some M with $\Gamma \vdash L <: M$.

References

[1] Benjamin Pierce, *Types and Programming Languages*, The MIT Press, Cambridge, Massachusetts London, England, 2002.

[2] Benjamin Pierce, Martin Steffen, *Higher-order Subtyping*, Theoretical Computer Science 176 (1997) 235-282.

[3] Dulma Rodriguez, Andreas Abel, *Algorithmic subtyping for higher-order bounded quantification*, Department of Computer Science, University of Munich.

[4] Henk Barendregt, *Lambda Calculi with Types*, Oxford University Press, USA (March 18, 1993)

[5] Morten Sorensen, Paweł Urzyczyn, *Lectures on the Curry-Howard Isomorphism*, Elsevier Science & Technology 2006