Expansion for Universal Quantifiers

Sergueï Lenglet, J Wells

Expansion for Universal Quantifiers

Sergueï Lenglet[1] ⋆ and J. B. Wells[2]

[1] University of Wrocław
[2] Heriot-Watt University

Abstract. *Expansion* is an operation on typings (i.e., pairs of typing environments and result types) defined originally in type systems for the λ-calculus with intersection types in order to obtain principal (i.e., most informative, strongest) typings. In a type inference scenario, expansion allows postponing choices for whether and how to use non-syntax-driven typing rules (e.g., intersection introduction) until enough information has been gathered to make the right decision. Furthermore, these choices can be equivalent to inserting uses of such typing rules at deeply nested positions in a typing derivation, without needing to actually inspect or modify (or even have) the typing derivation. Expansion has in recent years become simpler due to the use of *expansion variables* (e.g., in System E). This paper extends expansion and expansion variables to systems with ∀-quantifiers. We present System F_s, an extension of System F with expansion, and prove its main properties. This system turns type inference into a constraint solving problem; this could be helpful to design a modular type inference algorithm for System F types in the future.

1 Introduction

1.1 Background and Motivation

Polymorphism and principal typings. Many practical uses of type systems require *polymorphism*, i.e., the possibility to reuse a generic piece of code with different types. Type systems most commonly provide polymorphism through ∀-*quantifiers*, like in the Hindley-Milner (HM) type system [16] and in System F [19, 7], but can also use other methods like *intersection types* [3]. Systems with ∀-quantifiers assign general type schemes that can be instantiated to more specific types; for example, the identity function can be typed with $\forall a.(a \to a)$, and then used with types $\mathsf{int} \to \mathsf{int}$ or $\mathsf{real} \to \mathsf{real}$ when applied respectively to an integer or a real. Systems with intersection types list the different usage types of a term; if the identity function is applied exactly twice in a code fragment, once to an integer and once to a real, then its type will be $(\mathsf{int} \to \mathsf{int}) \cap (\mathsf{real} \to \mathsf{real})$.

Type systems with ∀-quantifiers are very popular, but they often lack *principal typings* [26], i.e., strongest, most informative typings (a typing is usually a pair of a type environments and a result type). Wells [26] proved that HM and System F do not have principal typings. It is important not to confuse this notion with the (weaker) one of "principal types" defined for the HM type system

⋆ The author is supported by the Alain Bensoussan Fellowship Programme

in which typable terms admit a strongest result type for each fixed type environment. Principal typings are crucial for *compositional* type inference, where types for terms are found using only the analysis results of the immediate subcomponents, which can be inspected independently and in any order. Compositional type inference helps in performing separate analysis of program modules, and therefore helps with separate compilation. Note that the Damas-Milner algorithm [4] for HM is not fully compositional: to give a type for a let-binding $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, the algorithm must infer first a type for e_1, and then use the result to type e_2.

Expansion and expansion variables. In contrast, type systems with intersection types usually have principal typings [3]. In such systems, admissible typings are obtained from a principal one using *expansion* (in addition to substitution and weakening). We present this mechanism through an example, taken from [2]. Consider the following λ-terms:

$$M_1 = \lambda x.x\ (\lambda y.y\ z) \qquad M_2 = \lambda g.\lambda x.g\ (g\ x)$$

One can compute the following principal typings for these terms in the type system of Coppo, Dezani, and Veneri [3].

$$M_1 : \langle z : a \vdash \underbrace{(((a \to b) \to b) \to c)}_{T_1} \to c \rangle$$
$$M_2 : \langle \emptyset \vdash \underbrace{((e \to f) \cap (d \to e)) \to (d \to f)}_{T_2} \rangle$$

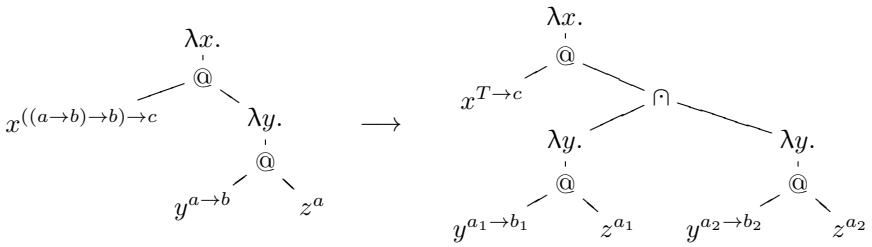
Following [2], we write $M : \langle A \vdash T \rangle$ for the assignment of type T under type environment A (often written $A \vdash M : T$ in the literature). To type the application $M_1\ M_2$, we must somehow "unify" T_1 and T_2. We cannot do this by simple type substitutions, replacing type variables by types; we have a clash between type $(a \to b) \to b$ and type $(e \to f) \cap (d \to e)$. We cannot unify these types by removing the intersection, using idempotence $T \cap T = T$; we would have to solve the equation $a \to b = b$, which does not have a solution in absence of recursive types.

This inference problem can be solved by introducing an intersection in the typing of M_1, using *expansion*.

$$M_1 : \langle z : a_1 \cap a_2 \vdash (((a_1 \to b_1) \to b_1 \cap (a_2 \to b_2) \to b_2) \to c) \to c \rangle$$

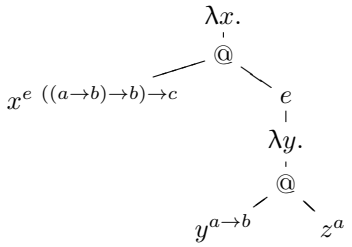
We can then unify the two types as required by applying the substitution $e := a_1 \to b_1, f := b_1, d := a_2 \to a_1 \to b_1, b_2 := a_1 \to b_1, c := (a_2 \to a_1 \to b_1) \to b_1$

The expansion operation simulates on typings the use of an intersection introduction typing rule at a nested position in the typing derivation. The above expansion on the typing of M_1 transforms the typing derivation on the left in the figure below into the derivation on the right (we write @ for the application typing rule, λ and $\cap$ for respectively abstraction and intersection introductions),



where $T = ((a_1 \to b_1) \to b_1) \cap ((a_2 \to b_2) \to b_2)$.

Earlier definitions of expansion [3, 20] are quite difficult to follow and to implement. *Expansion variables* (or E-variables) were introduced by Kfoury and Wells in System I [8] to simplify expansion application. The construct has then been improved in System E [1]. An E-variable e is a placeholder for unknown uses of typing rules such as $\cap$-introduction. For example, the following typing derivation for M_1



generates this typing:

$$M_1 : \langle z : e\ a \vdash (e\ ((a \to b) \to b) \to c) \to c \rangle$$

Note that the variable e is introduced in the result type as well as in the type environment. One can then perform the previous expansion by replacing e by the *expansion term* $(a := a_1, b := b_1) \cap (a := a_2, b := b_2)$, which introduces an intersection $\cap$ at the e position and applies a different substitution for each branch of the intersection. We then obtain the desired typing with intersection, given above.

Motivation. The idea behind expansion is fairly general, even if it has been defined only in systems with intersection types. It allows postponing the uses of non-syntactic typing rules, i.e., rules that are not driven by the syntax of terms, such as $\cap$-introduction, but also $\forall$-introduction and $\forall$-elimination. This is helpful in type inference scenarios: constructor introductions or eliminations can be delayed until all necessary information has been gathered. In the above example, we introduce an intersection in the typing of M_1 only when we have to, when applying M_1 to M_2. We want to bring this possibility of delaying the choice of uses of typing rules to type system with $\forall$-quantifiers, to see how (compositional) type inference could benefit from this property. We present an extension of System F with an expansion mechanism, called System $\mathrm{F_s}$. Before

going into the details of its syntax in Section 2, we first informally introduce System $\mathrm{F_s}$ and point out the main differences between its expansion mechanism and the one of System E.

1.2 Overview of System $\mathbf{F_s}$

Quantifier introduction. Assume that we have the following typings for the terms M_1 and M_2 given above.

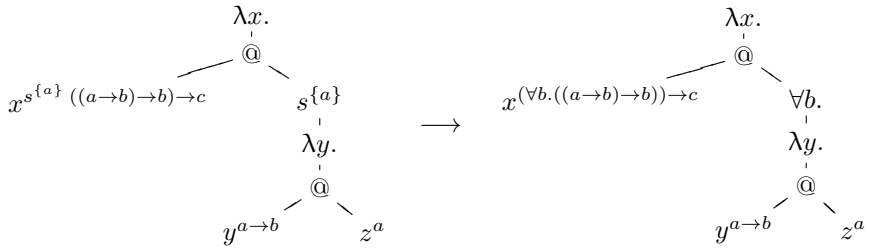
$$M_1 : \langle z : a \vdash \underbrace{(((a \to b) \to b) \to c)}_{T_1} \to c \rangle$$

$$M_2 : \langle \emptyset \vdash \underbrace{(\forall e.((d \to e) \to e)) \to (d \to d \to f) \to f}_{T_2} \rangle$$

Suppose we have forgotten M_1 and M_2 (e.g., we have already compiled them and discarded the source code), and we want to type the application $M_1\ M_2$. We need to "unify" T_1 and T_2. We cannot unify $(a \to b) \to b$ and $\forall e.((d \to e) \to e)$ using only type substitutions, because of the $\forall$-quantifier. This $\forall$-quantifier is necessary, because the term g is used twice in M_2 with different usage types. We can solve this problem by introducing in T_1 a $\forall$-quantifier over b, the scope of which encompasses $(a \to b) \to b$. To this end, we introduce an *expansion variable* s at the required position in the typing of M_1 (we use s instead of e to avoid confusion with the E-variables of System E).

$$M_1 : \langle z : a \vdash (s^{\{a\}}\ ((a \to b) \to b) \to c) \to c \rangle$$

Unlike expansion variables in System E, s is not introduced in the type environment; the application of s to the typing is asymmetric. We discuss the role of the superscript $\{a\}$ below. A $\forall$-quantifier over b can be introduced at the position we want by replacing s by the *expansion term* $\forall b$. This operation corresponds to the following transformation on derivation trees



and generates the typing

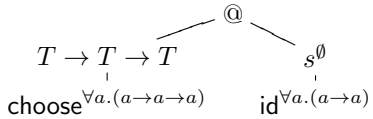
$$M_1 : \langle z : a \vdash (\forall b.((a \to b) \to b) \to c) \to c \rangle$$

as wished. We can then unify $\forall b.((a \to b) \to b) \to c$ with T_2, by substituting d for a and $(d \to d \to f) \to f$ for c. The key point is we can get the new typing without needing to build the typing derivation (or have any memory of M_1).

When we introduce a $\forall$-quantifier, we forbid any quantification over type variables that are free in the type environment. To take this into account, we keep the set of free variables of the environment as a parameter of the E-variable. For example, when we introduce s in the typing of M_1, a is the only free variable occurring in the environment; we remember the set $\{a\}$ in $s^{\{a\}}$. This prevents any illegal quantification from happening; replacing s by the expansion $\forall a$ does not introduce a quantification over a in this case and leaves the typing judgement unchanged.

Subtyping. E-variables can be used to perform subtyping as well. Consider System F $\forall$-elimination as a subtyping relation: $\forall a.T_1 \leq [a := T_2]T_1$. Let $A = \mathsf{choose} : \forall a.(a \to a \to a), \mathsf{id} : \forall a.(a \to a)$ and suppose we want to type the application $M = \mathsf{choose}\ \mathsf{id}$ under A (this example is taken from [11]). We can derive the typing $\langle A \vdash (\forall a.(a \to a)) \to (\forall a.(a \to a))\rangle$ for M; however if we want to apply M to a term of type $b \to b$, we have to redo the type inference on M to obtain the needed typing $\langle A \vdash (b \to b) \to (b \to b)\rangle$.

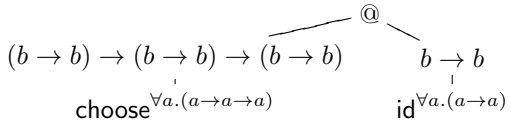
To avoid this, we add an E-variable s on top of the type of id; we obtain the following typing derivation (nodes marked with a type represent uses of subtyping, i.e., in our case, instantiations of $\forall$-quantifiers)



with $T = s^{\emptyset}\ \forall a.(a \to a)$, giving typing

$$M : \langle A \vdash (s^{\emptyset}\ \forall a.(a \to a)) \to (s^{\emptyset}\ \forall a.(a \to a))\rangle$$

If we want to apply M to a term M' of type $b \to b$, we utilize expansion to introduce the use of subtyping $\forall a.(a \to a) \leq b \to b$ at the s position in the typing tree. In the process, the type $T \to T \to T$ is updated into $(b \to b) \to (b \to b) \to (b \to b)$. We obtain



with typing $M : \langle A \vdash (b \to b) \to (b \to b)\rangle$, and we can then type $M\ M'$. In fact, the expansion mechanism for subtyping introduction does not depend on the definition of $\leq$, and therefore we keep System $\mathrm{F_s}$ parametric in its subtyping relation.

1.3 Summary of contributions

We define System $\mathrm{F_s}$ and present its principal properties. Improvements over previous work are as follows:

$$
\begin{array}{rlll}
x \in \mathsf{TermVar} & ::= \mathsf{x}_i & M \in \mathsf{Term} & ::= x \mid \lambda x.M \mid M_1 \mathbin{@} M_2 \\
a, b \in \mathsf{TypeVar} & ::= \mathsf{a}_i & T \in \mathsf{Type} & ::= a \mid T_1 \to T_2 \mid \forall a.T \mid s^B\ T \\
s \in \mathsf{ExpVar} & ::= \mathsf{s}_i & S \in \mathsf{Substitution} & ::= a := T, S \mid s := L, S \mid \boxdot \\
B \in \mathcal{P}_{\mathsf{fin}}(\mathsf{TypeVar}) & & L \in \mathsf{Expansion} & ::= \diamond \mid \forall a.L \mid s^B\ L \mid L^{:T} \\
 & & \Delta \in \mathsf{Constraint} & ::= \top \mid T_1 \lessdot T_2 \mid \Delta_1 \wedge \Delta_2 \mid \exists a.\Delta \mid s^B_T\ \Delta \\
 & & A \in \mathsf{TypeEnv} & ::= \emptyset \mid A, x : T \\
 & & Q \in \mathsf{Skeleton} & ::= x^A \mid \lambda x.Q \mid Q_1 \mathbin{@} Q_2 \mid \forall a.Q \mid s^B\ Q \mid Q^{:T}
\end{array}
$$

Fig. 1: Syntax grammars and metavariable conventions

1. System $\mathrm{F_s}$ is the first type system with an expansion mechanism for $\forall$-quantifiers, where we can delay $\forall$-introduction and uses of subtyping with expansion.
2. System $\mathrm{F_s}$ extends the notion of expansion; we introduce a new expansion mechanism with its corresponding (asymmetric) E-variables, which differ greatly from the ones of System E [1].
3. We prove that we can generate all System $\mathrm{F_s}$ judgements from a *initial skeleton*, an incomplete typing derivation with constraints that need to be solved. This property is a (weaker) form of principality (Theorem 5.4).
4. System $\mathrm{F_s}$ is parametric in its subtyping relation; by using different subtyping relations (such as System F type application or Mitchell's relation [17]), one can change the typing power of System $\mathrm{F_s}$ without modifying the typing rules or judgements.
5. System $\mathrm{F_s}$ turns type inference into a type constraint solving problem. We believe it can be helpful to reason about modular type inference, even if we do not provide a constraint solving algorithm yet.

Proofs are available in an accompying research report [14].

2 Syntax

Fig. 1 defines the grammars and metavariable conventions of the entities used in this paper. Let i, j, m, n range over natural numbers. Given a set X, we write $\mathcal{P}_{\mathsf{fin}}(X)$ for the set of finite subsets of X. We distinguish between the metavariables x, a, s, and the concrete variables x_i, a_i, s_i. The (non-standard) symbol @ used for application helps in reading skeletons, and we keep it for terms for consistency. We explain the role of constraints (Δ) and skeletons (Q) in Section 3, and the syntax of expansion terms (L) and substitutions (S) in Section 4.

Precedence. To reduce parenthesis usage, we define precedence for operators and operations defined later (such as substitution and expansion applications $[S]T$ and $[\![L]\!]^B T$) in the following order, from highest to lowest: $s^B\ T$, $\forall a.T$, $[S]T$, $[\![L]\!]^B T$, $T_1 \to T_2$. For example, $[S]T_1 \to s^B\ T_2 = ([S]T_1) \to (s^B\ T_2)$ and $\forall a.a \to \forall a.a = (\forall a.a) \to (\forall a.a)$. Furthermore, the function type constructor is

$$x^A \triangleright x : \langle A \vdash A(x) \rangle / \top \quad (\mathsf{var}) \qquad \frac{Q \triangleright M : \langle A, x : T_1 \vdash T_2 \rangle / \Delta}{\lambda x.Q \triangleright \lambda x.M : \langle A \vdash T_1 \to T_2 \rangle / \Delta} \quad (\mathsf{abs})$$

$$\frac{Q_1 \triangleright M_1 : \langle A \vdash T_1 \to T_2 \rangle / \Delta_1 \qquad Q_2 \triangleright M_2 : \langle A \vdash T_1 \rangle / \Delta_2}{Q_1 \mathbin{@} Q_2 \triangleright M_1 \mathbin{@} M_2 : \langle A \vdash T_2 \rangle / (\Delta_1 \wedge \Delta_2)} \quad (\mathsf{app})$$

$$\frac{Q \triangleright M : \langle A \vdash T \rangle / \Delta \qquad a \notin \mathsf{ftv}(A)}{\forall a.Q \triangleright M : \langle A \vdash \forall a.T \rangle / \exists a.\Delta} \quad (\forall\text{-I}) \qquad \frac{Q \triangleright M : \langle A \vdash T \rangle / \Delta \qquad \mathsf{ftv}(A) \subseteq B}{s^B\, Q \triangleright M : \langle A \vdash s^B\, T \rangle / s_T^B\, \Delta} \quad (s\text{-I})$$

$$\frac{Q \triangleright M : \langle A \vdash T_1 \rangle / \Delta}{Q^{:T_2} \triangleright M : \langle A \vdash T_2 \rangle / (\Delta \wedge (T_1 \lessdot T_2))} \quad (\lessdot)$$

Fig. 2: Typing rules

right-associative, so that $T_1 \to T_2 \to T_3 = T_1 \to (T_2 \to T_3)$, and the application is left-associative, so that $M_1 \mathbin{@} M_2 \mathbin{@} M_3 = (M_1 \mathbin{@} M_2) \mathbin{@} M_3$.

Equalities and α-conversion. We allow α-conversion of bound variables in types (where $\forall a.T$ binds a), skeletons (where $\lambda x.Q$ binds x and $\forall a.Q$ binds a), and constraints (where $\exists a.\Delta$ binds a). Note that a is not bound in the expansion term $\forall a.L$, and therefore it cannot be α-converted.

We equate types up to reordering of adjacent $\forall$-quantifiers (so $\forall a_1.\forall a_2.{}_2T = \forall a_2.\forall a_1.T$), and suppression of dummy quantifiers (if a is not free in T, then $\forall a.T = T$). We also enforce the following equalities on constraints

$$\begin{array}{lll} \exists a.(\Delta_1 \wedge \Delta_2) = (\exists a.\Delta_1) \wedge (\exists a.\Delta_2) & \Delta \wedge \Delta = \Delta & \Delta \wedge \top = \Delta \\ s_T^B\, (\Delta_1 \wedge \Delta_2) = (s_T^B\, \Delta_1) \wedge (s_T^B\, \Delta_2) & \Delta_1 \wedge \Delta_2 = \Delta_2 \wedge \Delta_1 & \\ \Delta_1 \wedge (\Delta_2 \wedge \Delta_3) = (\Delta_1 \wedge \Delta_2) \wedge \Delta_3 & \exists a.\Delta = \Delta \text{ if } a \text{ is not free in } \Delta & \end{array}$$

Auxiliary notations and functions. Let $\mathsf{fv}(M)$ be the set of free variables of M, defined in the usual way. The free type variables of a type, an expansion, and a substitution are defined as follows.

$$\begin{array}{llll} \mathsf{ftv}(a) & = \{a\} & \mathsf{ftv}(\diamond) & = \emptyset \\ \mathsf{ftv}(T_1 \to T_2) & = \mathsf{ftv}(T_1) \cup \mathsf{ftv}(T_2) & \mathsf{ftv}(L^{:T}) & = \mathsf{ftv}(L) \cup \mathsf{ftv}(T) \\ \mathsf{ftv}(\forall a.T) & = \mathsf{ftv}(T) \setminus \{a\} & \mathsf{ftv}(\forall a.L) & = \mathsf{ftv}(L) \cup \{a\} \\ \mathsf{ftv}(s^B\, T) & = \mathsf{ftv}(T) \cup B & \mathsf{ftv}(s^B\, L) & = \mathsf{ftv}(L) \cup B \end{array}$$

$$\begin{array}{ll} \mathsf{ftv}(\boxdot) & = \emptyset \\ \mathsf{ftv}(a := T, S) & = \{a\} \cup \mathsf{ftv}(T) \cup \mathsf{ftv}(S) \\ \mathsf{ftv}(s := L, S) & = \mathsf{ftv}(L) \cup \mathsf{ftv}(S) \end{array}$$

3 Typing rules

A type environment A (defined in Fig. 1) is a list of assignments which maps term variables to types. When writing a non-empty environment, we allow omitting

the leading symbols "$\emptyset$,". A type environment is *well-formed* iff it does not mention twice the same term variable. Henceforth, we consider only well-formed type environments. For $A = x_1 : T_1, \ldots, x_n : T_n$, we define $A(x_i) = T_i$ for $i \in \{1 \ldots n\}$, $\mathsf{ftv}(A) = \bigcup_{i \in \{1 \ldots n\}} \mathsf{ftv}(T_i)$, and $\mathsf{support}(A) = \{x_1 \ldots x_n\}$.

The typing rules of System $\mathrm{F_s}$ (Fig. 2) derive judgements of the form $Q \triangleright M : \langle A \vdash T \rangle / \Delta$, where constraints that need to be solved (by type inference) are accumulated in Δ. A constraint of the form $T_1 \lessdot T_2$ is called *atomic*. By including constraints in judgements, we can use the same rules for type checking and type inference. If the constraint is *solved* w.r.t. some subtyping relation, then the judgement acts as a regular typing judgement, assigning *typing* $\langle A \vdash T \rangle$ to the untyped term M.

A skeleton Q is just a *proof term*, a compact piece of syntax which represents a complete typing derivation. A skeleton Q is *valid* iff there exist M, A, T, and Δ such that $Q \triangleright M : \langle A \vdash T \rangle / \Delta$. Henceforth, we consider only valid skeletons. All components of a judgement $Q \triangleright M : \langle A \vdash T \rangle / \Delta$ are uniquely determined by Q, therefore we can define functions rtype and tenv such that $\mathsf{rtype}(Q) = T$ and $\mathsf{tenv}(Q) = A$. Skeletons replace typing derivation trees in formal statements. For example, $\lambda x.(x^{x:\forall a.a})^{:(\forall a.a) \to b} \mathbin{@} x^{x:\forall a.a}$ represents the following derivation.

$$\dfrac{\dfrac{x : \langle x : \forall a.a \vdash \forall a.a \rangle / \top}{x : \langle x : \forall a.a \vdash (\forall a.a) \to b \rangle / (\forall a.a \lessdot (\forall a.a) \to b)} \qquad x : \langle x : \forall a.a \vdash \forall a.a \rangle / \top}{x \mathbin{@} x : \langle x : \forall a.a \vdash b \rangle / (\forall a.a \lessdot (\forall a.a) \to b)}}{\lambda x.x \mathbin{@} x : \langle \emptyset \vdash (\forall a.a) \to b \rangle / (\forall a.a \lessdot (\forall a.a) \to b)}$$

In examples, we sometimes omit skeletons and constraints when they are not relevant, writing $M : \langle A \vdash T \rangle$ iff there exists Q, Δ such that $Q \triangleright M : \langle A \vdash T \rangle / \Delta$.

Remark 3.1. A variable skeleton x^A remembers a type environment A and not simply the type of x to be able to type a variable x in a term $\lambda x.M$ such that $x \notin \mathsf{fv}(M)$. For example, we have $\lambda x.y^{x:a,y:b} \triangleright \lambda x.y : \langle y : b \vdash a \to b \rangle / \top$.

We could have used λ-terms with only type annotations on bindings, like many other systems, but our skeletons are also useful because they uniquely represent entire typing derivations (judgement trees). We also prefer our skeletons because a goal for future work is a system containing both System E and System $\mathrm{F_s}$ (cf. Section 8), and our format of skeleton is better suited for the intersection introduction typing rule of System E, as discussed in [27]. □

Rules (var), (abs), and (app) are classic. The subtyping rule ($\lessdot$) generates a new atomic constraint, the meaning of which depends on the chosen subtyping relation (cf. solvedness definition in Section 6.1). Rule (∀-I) introduces a ∀-quantifier over a, provided that a is not free in A. Note that a may occur free in Δ; we use an existential quantifier $\exists a.\Delta$ to bind it, as solvedness requires Δ to be solved for some a (cf. Section 6.1), and not for all possible instantiations of a, as a ∀-binder would suggest.

Rule (s-I) introduces an expansion variable s to mark a position in the derivation tree where a ∀-quantifier can be added or where subtyping can be used.

$$
\begin{array}{ll}
[\![\diamond]\!]^B W = W & [\![\diamond]\!]^B_T \Delta = \Delta \\
[\![s^{B'}\ L]\!]^B W = s^{B \cup B'}\ ([\![L]\!]^{B'} W) & [\![s^{B'}\ L]\!]^B_T \Delta = s^{B \cup B'}_{[\![L]\!]^B T}\ ([\![L]\!]^B_T \Delta) \\
[\![\forall a.L]\!]^B W = \begin{cases} \forall a.[\![L]\!]^B W \text{ if } a \notin B \\ [\![L]\!]^B W \quad \text{otherwise} \end{cases} & [\![\forall a.L]\!]^B_T \Delta = \begin{cases} \exists a.[\![L]\!]^B_T \Delta \text{ if } a \notin B \\ [\![L]\!]^B_T \Delta \quad \text{otherwise} \end{cases} \\
[\![L^{:T_2}]\!]^B T_1 = T_2 & [\![L^{:T_2}]\!]^B_{T_1} \Delta = ([\![L]\!]^B_{T_1} \Delta) \wedge (([\![L]\!]^B T_1) \lessdot T_2) \\
[\![L^{:T_2}]\!]^B Q = ([\![L]\!]^B Q)^{:T_2} &
\end{array}
$$

Fig. 3: Expansion application

Because a quantification over a free variable of A is not allowed (rule $(\forall\text{-I})$), the E-variable remembers an over-approximation B of $\mathsf{ftv}(A)$, which is used by the expansion mechanism to prevent any illegal $\forall$-introduction from happening. The type T mentioned in $s^B_T\ \Delta$ can be used during expansion to generate an atomic constraint $T \lessdot T'$ if needed. We explain the expansion mechanism in detail in the next section.

Remark 3.2. The rule (var) may also introduce E-variables, as for example in $x^{x:s^\emptyset\ a} \triangleright x : \langle x : s^\emptyset\ a \vdash s^\emptyset\ a \rangle / \top$. In this case, performing expansion at the position of s does not correspond to a use of rules $(\forall\text{-I})$ or $(\lessdot)$, and the set B of type variables remembered by s can be any set. Indeed we can derive $x^{x:s^B\ a} \triangleright x : \langle x : s^B\ a \vdash s^B\ a \rangle / \top$ for any B. □

Remark 3.3. In rule $(s\text{-I})$, we can remember a set bigger than $\mathsf{ftv}(A)$ for subject reduction to hold. For example, consider the following judgement

$$Q \triangleright (\lambda x.y)\,@\,\lambda x.x : \langle y : b \vdash s^{\{a,b\}}\ b \rangle / s^{\{a,b\}}_b\ \top$$

with $Q = (\lambda x.s^{\{a,b\}}\ y^{x:a \to a, y:b})\,@\,\lambda x.x^{x:a,y:b}$. The term $(\lambda x.y)\,@\,\lambda x.x$ reduces to y, and to derive

$$s^{\{a,b\}}\ y^{y:b} \triangleright y : \langle y : b \vdash s^{\{a,b\}}\ b \rangle / s^{\{a,b\}}_b\ \top,$$

we have to be able to mention a even if it does not appear in $y : b$. □

4 Substitution and expansion

4.1 Expansion application

The syntax of expansion terms is given in Fig. 1. Let W range over types and skeletons. Fig. 3 defines the application of expansion to types, skeletons, and constraints. When applied to a type or a skeleton, the expansion mechanism relies on a set of type variables B, used in introductions of E-variable and $\forall$-quantifier; when applied to a constraint, it requires an extra parameter (a type) to generate an appropriate atomic constraint if needed. Each construct of expansion terms

Metavariables

$v ::= a \mid s$

$\Phi ::= T \mid L$

Substitution application

$$
\begin{array}{ll}
[\boxdot]a & = a \\
[\boxdot]s & = s^{\emptyset}\ \diamondsuit \\
[v := \Phi, S]v & = \Phi \\
[v := \Phi, S]v' & = [S]v' \text{ if } v \neq v' \\
[S](s^B\ T) & = [\![[S]s]\!]^{\mathsf{ftv}([S]B)}[S]T \\
[S]\forall a.T & = \forall a.[S]T \text{ if } a \notin \mathsf{ftv}(S) \\
[S](T_1 \to T_2) & = [S]T_1 \to [S]T_2
\end{array}
$$

$$
\begin{array}{ll}
[S]x^A & = x^{[S]A} \\
[S]\lambda x.Q & = \lambda x.[S]Q \\
[S](Q_1 \mathbin{@} Q_2) & = ([S]Q_1) \mathbin{@} ([S]Q_2) \\
[S](s^B\ Q) & = [\![[S]s]\!]^{\mathsf{ftv}([S]B)}[S]Q \\
[S]\forall a.Q & = \forall a.[S]Q \text{ if } a \notin \mathsf{ftv}(S) \\
[S](Q^{:T}) & = [S]Q^{:[S]T} \\
 & \\
[S](T_1 \lessdot T_2) & = [S]T_1 \lessdot [S]T_2 \\
[S]\top & = \top \\
[S](s^B_T\ \Delta) & = [\![[S]s]\!]^{\mathsf{ftv}([S]B)}_{[S]T}[S]\Delta \\
[S]\exists a.\Delta & = \exists a.[S]\Delta \text{ if } a \notin \mathsf{ftv}(S) \\
[S](\Delta_1 \wedge \Delta_2) & = ([S]\Delta_1) \wedge ([S]\Delta_2)
\end{array}
$$

Fig. 4: Substitution application

corresponds to the application of a non-syntactic typing rule, except for the null expansion $\diamondsuit$, which leaves unchanged the entities it is applied to.

E-variable and $\forall$-quantifier expansions behave the same on types, skeletons, and constraints. Applied with parameter B, the expansions $s^{B'}\ L$ and $\forall a.L$ first execute L and then introduce an E-variable s (with set $B \cup B'$ of variables that cannot be quantified) and a quantifier over a (iff $a \notin B$), respectively. When applied to all parts of a judgement $Q \triangleright M : \langle A \vdash T \rangle / \Delta$, we must have $\mathsf{ftv}(A) \subseteq B$ for these operations to be sound w.r.t. rules (s-I) and ($\forall$-I) (cf. Lemma 4.1).

The expansion $L^{:T_2}$ first applies L and then performs subtyping with T_2, as we can see in the skeleton case. When applied to a type, only the subtyping step matters, and we simply obtain T_2. Finally, the constraint case Δ requires an extra parameter T_1 to generate a new atomic constraint. In practice, T_1 will be the result type of the judgement $Q \triangleright M : \langle A \vdash T_1 \rangle / \Delta$ from which Δ comes. When $L^{:T_2}$ is applied to the above judgement, L is applied first, in particular to the type T_1. To take this into account, the generated constraint is $([\![L]\!]^B T_1) \lessdot T_2$ (and not simply $T_1 \lessdot T_2$).

Expansion is sound w.r.t. to the type system of System $\mathrm{F_s}$.

Lemma 4.1. *If* $Q \triangleright M : \langle A \vdash T \rangle / \Delta$ *and* $\mathsf{ftv}(A) \subseteq B$*, then* $[\![L]\!]^B Q \triangleright M : \langle A \vdash [\![L]\!]^B T \rangle / [\![L]\!]^B_T \Delta$. □

Expansion operates only at the top-level of the typing judgement in Lemma 4.1; in order to expand at a deeply nested position, we have to replace an E-variable s by an expansion L, as explained in the next section.

4.2 Substitution application

Substitutions (defined in Fig. 1) are lists of assignments that map type variables to types ($a := T$) and E-variables to expansions ($s := L$), ended by the symbol $\boxdot$. Application of substitutions to type variable sets B and type environments A

is pointwise. Given a finite set of types $\{T_1 \dots T_n\}$, we define $\mathsf{ftv}(\{T_1 \dots T_n\})$ as $\bigcup_{i \in \{1 \dots n\}} \mathsf{ftv}(T_i)$. Fig. 4 defines application of substitutions to variables, types, skeletons, and constraints.

A substitution S generates a type T (resp. an expansion L) when applied to a type variable a (resp. to an E-variable s). A substitution may contain several assignments for the same variable, as in $S = (a := T_1, a := T_2, \boxdot)$; in this case, only the first one is considered. We choose this design for simplicity; an alternate solution would be to syntactically prevent repetitions in the substitution definition, but the definition would then become more complex for no obvious gain.

The application of substitutions to types $s^B\ T$ is the most important case.

$$[S](s^B\ T) = [\![[S]s]\!]^{\mathsf{ftv}([S]B)}[S]T$$

The substitution S is first applied to s, which gives us an expansion $L = [S]s$, which is then applied to the type $[S]T$. We remember that B is (an over-approximation of) the set of free type variables that cannot be quantified over, because they appear in the type environment at the time the variable s is introduced. If S replaces a variable $a \in B$ by a type T', then T' now appears in the type environment, and its free variables cannot be quantified over. This explains why we have to apply the expansion $[\![[S]s]\!]^{\mathsf{ftv}([S]B)}[S]T$ with the set $\mathsf{ftv}([S]B)$ and not simply with the set B. The application of S to skeletons $s^B\ Q$ and to constraints $s^B_T\ \Delta$ follows the same pattern.

Example 4.2. Let $M = \lambda x.x \mathbin{@} y$. We have

$$M : \langle y : a \vdash s^{\{a\}}\ ((a \to b) \to b) \rangle$$

Applying $S_1 = (a := a_1 \to a_2, \boxdot)$ to this typing gives us

$$M : \langle y : a_1 \to a_2 \vdash s^{\{a_1, a_2\}}\ (((a_1 \to a_2) \to b) \to b) \rangle$$

Then applying $S_2 = (s := \forall b.\diamondsuit, \boxdot)$ gives us

$$M : \langle y : a_1 \to a_2 \vdash \forall b.(((a_1 \to a_2) \to b) \to b) \rangle$$

Note that the substitution $(s := \forall a'.\diamondsuit, \boxdot)$ would have left the last judgement unchanged if $a' \in \{a_1, a_2\}$, and would have introduced a dummy quantifier if $a' \notin \{b, a_1, a_2\}$. We can achieve the same effect as doing S_1 before S_2 by applying the substitution $S = (a := a_1 \to a_2, s := \forall b.\diamondsuit, \boxdot)$ to the initial judgement. □

Example 4.3. Let $T = \forall a.(a \to a)$. We have

$$\lambda x.s^{\emptyset}\ ((x^{x:T})^{:T \to T} \mathbin{@} x^{x:T}) \triangleright \lambda x.x \mathbin{@} x : \langle \emptyset \vdash T \to s^{\emptyset}\ T \rangle / s^{\emptyset}_T\ (T \lessdot T \to T)$$

Applying substitution $S = (s := \diamondsuit^{:b \to b}, \boxdot)$ gives us

$$\lambda x.((x^{x:T})^{:T \to T} \mathbin{@} x^{x:T})^{:b \to b} \triangleright \lambda x.x \mathbin{@} x : \langle \emptyset \vdash T \to b \to b \rangle / \Delta$$

where $\Delta = (T \lessdot b \to b) \wedge (T \lessdot T \to T)$. Subtyping has been introduced at a nested position (under the λ), generating the expected constraint $T \lessdot b \to b$. □

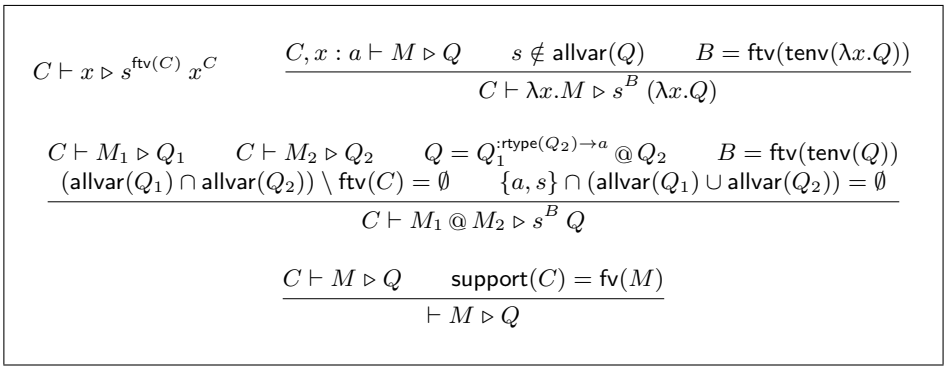


Fig. 5: Initial skeletons of a term

Substituting a variable s by an expansion L makes s disappear. As a result, one can use the null expansion $\diamond$ to delete an E-variable s from a type $s^B\,T$. If $S = (s := \diamond, \boxdot)$, then $[S](s^B\,T) = [\![\diamond]\!]^B[S]T = [S]T$ (the occurrences of s in T are also removed). An expansion L can be applied at the location of a variable s without making s disappear using the substitution $S = (s := s^\emptyset\,L, \boxdot)$. Indeed we have $[S](s^B\,T) = [\![s^\emptyset\,L]\!]^B[S]T = s^B\,[\![L]\!]^B[S]T$. The substitution $\boxdot$ is the *identity* substitution; it leaves variables, types, skeletons, and constraints unchanged. For example, for E-variables, we have $[\boxdot](s^B\,T) = [\![s^\emptyset\ \diamond]\!]^B[\boxdot]T = s^B\,[\boxdot]T$. The remaining cases of substitution application are straightforward descending cases. The resulting operation is sound w.r.t. System $\mathrm{F_s}$ type system.

Theorem 4.4. *If* $Q \triangleright M : \langle A \vdash T\rangle/\Delta$ *then* $[S]Q \triangleright M : \langle [S]A \vdash [S]T\rangle/[S]\Delta$. $\square$

5 Initial Skeletons

In this section, we prove that we can generate all System $\mathrm{F_s}$ judgements for a term M from an initial skeleton built from M.

We first show that we can obtain *relevant* skeletons; a skeleton Q such that $Q \triangleright M : \langle A \vdash T\rangle/\Delta$ is relevant if $\mathsf{fv}(M) = \mathsf{support}(A)$. In words, the type environment of a relevant skeleton does not mention more term variables than necessary. A *variable environment* C is a type environment which assigns type variables to expression variables and such that for all x, y such that $x \neq y$, we have $C(x) \neq C(y)$. We write $\mathsf{allvar}(Q)$ for the set of free type and E-variables occurring in Q. Fig. 5 defines a judgement $\vdash M \triangleright Q$, which means that Q is an *initial skeleton* for M. The main ideas behind this construct are as follows: first, we type each variable in $\mathsf{fv}(M)$ with a distinct type variable (using the environment C mentioned in the auxiliary judgement $C \vdash M \triangleright Q$). Then we introduce a (fresh) E-variable at every possible position in the skeleton. Finally, we use subtyping to ensure that a term in a function position in an application has an arrow type. Two initial skeletons for the same term are equivalent up to renaming of their variables, as stated in the lemma below (where we call an expansion of the form $s^B\ \diamond$ an *E-expansion*).

Lemma 5.1. *Let Q_1, Q_2 such that $\vdash M \triangleright Q_1$ and $\vdash M \triangleright Q_2$. There exists a substitution S which maps type variables to type variables and E-variables to E-expansions such that $Q_1 = [S]Q_2$.* □

Example 5.2. Let $M = \lambda x.x \mathbin{@} x$. Then

$$Q = \mathsf{s}_3^{\emptyset}\, \lambda x.\mathsf{s}_2^{\{\mathsf{a}_0\}}\, ((\mathsf{s}_0^{\{\mathsf{a}_0\}}\, x^{x:\mathsf{a}_0})^{:(\mathsf{s}_1^{\{\mathsf{a}_0\}}\, \mathsf{a}_0 \to \mathsf{a}_1)} \mathbin{@} \mathsf{s}_1^{\{\mathsf{a}_0\}}\, x^{x:\mathsf{a}_0})$$

is an initial skeleton for M and we have

$$Q \triangleright M : \langle \emptyset \vdash \mathsf{s}_3^{\emptyset}\, (\mathsf{a}_0 \to \mathsf{s}_2^{\{\mathsf{a}_0\}}\, \mathsf{a}_1) \rangle / \Delta$$

with $\Delta = \mathsf{s}_3{}^{\emptyset}_{\mathsf{a}_0 \to \mathsf{s}_2^{\{\mathsf{a}_0\}}\, \mathsf{a}_1}\, \mathsf{s}_2{}^{\{\mathsf{a}_0\}}_{\mathsf{a}_1}\, ((\mathsf{s}_0^{\{\mathsf{a}_0\}}\, \mathsf{a}_0 \lessdot \mathsf{s}_1^{\{\mathsf{a}_0\}}\, \mathsf{a}_0 \to \mathsf{a}_1) \wedge \mathsf{s}_0{}^{\{\mathsf{a}_0\}}_{\mathsf{a}_0}\, \top \wedge \mathsf{s}_1{}^{\{\mathsf{a}_0\}}_{\mathsf{a}_0}\, \top)$. Roughly, the variables (s_i) can be used to introduce $\forall$-quantifiers or subtyping at their respective positions. For example, let $T = \forall a.(a \to a)$ and $S = (\mathsf{a}_0 := T, \mathsf{a}_1 := T, \mathsf{s}_0 := \diamondsuit, \mathsf{s}_1 := \diamondsuit, \mathsf{s}_2 := \diamondsuit^{:b \to b}, \mathsf{s}_3 := \forall b.\diamondsuit, \boxdot)$. Applying S to the above typing judgement, we obtain

$$\forall b.\lambda x.((x^{x:T})^{:T \to T} \mathbin{@} x^{x:T})^{:b \to b} \triangleright M : \langle \emptyset \vdash \forall b.(T \to b \to b) \rangle / [S]\Delta$$

with $[S]\Delta = \exists b.((T \lessdot T \to T) \wedge (T \lessdot b \to b))$. □

In the following, we use a predicate refl to check that a constraint is built from atomic constraints of the form $T \lessdot T$. The formal definition is

$$\mathsf{refl}(\top) \qquad \mathsf{refl}(T \lessdot T) \qquad \frac{\mathsf{refl}(\Delta)}{\mathsf{refl}(\exists a.\Delta)} \qquad \frac{\mathsf{refl}(\Delta)}{\mathsf{refl}(s_T^B\, \Delta)} \qquad \frac{\mathsf{refl}(\Delta_1) \qquad \mathsf{refl}(\Delta_2)}{\mathsf{refl}(\Delta_1 \wedge \Delta_2)}$$

A reflexive constraint is always solved w.r.t. a reflexive subtyping relation (see solvedness definition in the next section). From any initial skeleton of M, we can obtain all relevant skeletons for M.

Lemma 5.3. *Let $\vdash M \triangleright Q$. Let Q' relevant such that $Q' \triangleright M : \langle A \vdash T \rangle / \Delta$. There exists S such that $[S]Q \triangleright M : \langle A \vdash T \rangle / (\Delta \wedge \Delta')$ with Δ' reflexive.* □

Note that in the above lemma, we do not have $[S]Q = Q'$, and we obtain an approximation of Δ. By construction, an initial skeleton Q uses subtyping at each application node to generate an atomic constraint. Applying S turns these constraints into reflexive ones, but it cannot completely remove them. Therefore, $[S]Q$ is similar to Q' up to these uses of (reflexive) subtyping at application nodes.

To generate all possible typing derivations, we add a weakening rule to be able to extend a type environment.

$$\frac{Q \triangleright M : \langle A_1 \vdash T \rangle / \Delta \qquad \mathsf{support}(A_1) \cap \mathsf{support}(A_2) = \emptyset}{Q^{A_2} \triangleright M : \langle A_1, A_2 \vdash T \rangle / \Delta}$$

Theorem 5.4. *Let $\vdash M \triangleright Q$. If $Q' \triangleright M : \langle A \vdash T \rangle / \Delta$, then there exists S, A' such that $([S]Q)^{A'} \triangleright M : \langle A \vdash T \rangle / (\Delta \wedge \Delta')$, with Δ' reflexive.* □

We emphasize that initial skeletons are quite different from principal typings: initial skeletons are not typing derivations, because they contain unsolved constraints, and all terms, even non typable ones, have an initial skeleton. To obtain a principal typing from the initial skeleton, we need to solve the constraints in a principal manner; we conjecture that it is not possible, i.e., System $\mathrm{F_s}$ does not have principal typings, for the same reason as for System F [26].

Nevertheless, we think that initial skeletons can be useful for modular type inference. First, note that we do not have to remember the skeleton itself or the term; the typing and constraint contain all the information we need. Besides, constraint solving can be divided into solution preserving steps, which produce an equivalent constraint, and solution reducing steps, where some information is lost. It is always possible to safely perform solution preserving steps, and one can periodically check if it is possible to apply solution reducing steps to find at least one solved typing. The best intermediate representation might be a typing on which all known solution preserving steps have been performed, together with (at least) one solution reducing step of that typing's constraint. We do not know in practice how many steps will be solution preserving versus solution reducing.

An example use of System $\mathrm{F_s}$ is to look for a subsystem of System F in which to do compositional type inference. System $\mathrm{F_s}$ is a good framework in which to perform such a search, by considering various different restrictions of System $\mathrm{F_s}$ until one is found with the right properties. Because all possible System F derivations can be obtained from System $\mathrm{F_s}$ initial skeletons, we know in advance that the framework has the right amount of power. Such subsystems could also be characterized by a constraint solving algorithm. Instead of searching for a subsystem by varying the typing rules, we could vary the constraint solving algorithm, and when a nice algorithm is found, we could try to find a corresponding restriction directly stated on the typing rules.

6 Solvedness and Subject Reduction

6.1 Solvedness and System F

A constraint Δ is solved w.r.t. a subtyping relation $\leq$ if its atomic constraints are solved w.r.t. $\leq$. Formally, we define the predicate solved, as follows.

$$\mathsf{solved}(\top, \leq) \qquad \frac{T_1 \leq T_2}{\mathsf{solved}(T_1 \lessdot T_2, \leq)} \qquad \frac{\mathsf{solved}(\Delta_1, \leq) \qquad \mathsf{solved}(\Delta_2, \leq)}{\mathsf{solved}(\Delta_1 \wedge \Delta_2, \leq)}$$

$$\frac{\mathsf{solved}(\Delta, \leq)}{\mathsf{solved}(\exists a.\Delta, \leq)} \qquad \frac{\mathsf{solved}(\Delta, \leq)}{\mathsf{solved}(s_T^B\ \Delta, \leq)}$$

A skeleton is solved if its constraint is solved. Solved skeletons correspond to typing derivations in the traditional sense.

We can express System F in System $\mathrm{F_s}$ by using the following relation $\leq_{\mathrm{F}}$.

$$\forall a.T_1 \leq_{\mathrm{F}} [a := T_2, \boxdot]T_1 \quad (\forall\text{-}\mathsf{E})$$

Because of the equality involving dummy quantifiers, the relation $\leq$ is reflexive; indeed for $a \notin \mathsf{ftv}(T)$, we have $T = \forall a.T \leq_{\mathrm{F}} T$. Clearly, System F_{s} equipped with $\leq_{\mathrm{F}}$ extends System F. Conversely, it is easy to see that a term typable in System F_{s} is typable in F once we erase all the E-variables.

Proposition 6.1. *A term is typable in System F iff it is typable in System F_s with $\leq_{\mathrm{F}}$.* □

6.2 Subject Reduction

We now present the subject reduction result of System F_{s} with $\leq_{\mathrm{F}}$ w.r.t. call-by-value semantics. Let V range over values, i.e. $V ::= x \mid \lambda x.M$. We write $[x := M_1]M_2$ for the usual capture-avoiding substitution of terms. We define small-step call-by-value evaluation $M \xrightarrow{\mathsf{cbv}} M'$ as the smallest relation on terms verifying the following rules:

$$(\lambda x.M) \mathbin{@} V \xrightarrow{\mathsf{cbv}} [x := V]M \qquad \frac{M_1 \xrightarrow{\mathsf{cbv}} M_1'}{M_1 \mathbin{@} M_2 \xrightarrow{\mathsf{cbv}} M_1' \mathbin{@} M_2} \qquad \frac{M \xrightarrow{\mathsf{cbv}} M'}{V \mathbin{@} M \xrightarrow{\mathsf{cbv}} V \mathbin{@} M'}$$

Theorem 6.2. *If $Q \triangleright M : \langle A \vdash T \rangle / \Delta$, $\mathsf{solved}(\Delta, \leq_{\mathrm{F}})$, and $M \xrightarrow{\mathsf{cbv}} M'$, then there exists Q', Δ' such that $Q' \triangleright M' : \langle A \vdash T \rangle / \Delta'$ and $\mathsf{solved}(\Delta', \leq_{\mathrm{F}})$.* □

We prove Theorem 6.2 by defining a transformation on Q so that skeletons in a function position of an application, such as Q_1 in $Q_1 \mathbin{@} Q_2$, are turned into λ-abstraction skeletons. A substitution lemma then allows us to simulate β-reduction by replacing the occurrences of a variable skeleton x^A in a skeleton $\lambda x.Q_1'$ by Q_2. This proof technique depends on the subtyping relation being used. We conjecture it can be adapted to various relations (such as Mitchell's [17]), but nevertheless we look for a more generic proof technique (less dependant on the subtyping relation). We prove subject reduction only for call-by-value evaluation for simplicity; we conjecture that subject reduction also holds for call-by-need and call-by-name semantics, and for reduction in arbitrary contexts.

7 Related Work

7.1 Expansion

A full survey on expansion and expansion variables can be found in [2]; we only discuss here the main differences between System F_{s} and System E, the type system with expansion most closely related to our work. System E E-variables are introduced on top of skeletons, type environments, result types, and constraints, while System F_{s} E-variables are not inserted on top of type environments (rule (s-I)). System F_{s} expansion mechanism deals with subtyping, while System E expansion does not. In System E, an E-variable e defines a namespace. In type $T_1 = a \to e\ a$, the variable a outside e is not connected to the one in the scope of

e; applying substitution $(a := T_2, \boxdot)$ to T_1 gives $T_2 \to e\ a$. This is due to the fact that substitutions are a special case of System E expansions (see [2] for further details). It also makes composition of expansions and substitutions easier. In System $\mathrm{F_s}$, substitutions cannot be considered as expansions, because they are applied to the whole typing judgement (Theorem 4.4), whereas the asymmetric expansions of System $\mathrm{F_s}$ are not applied to the type environments (Lemma 4.1). As a result, it would be unsound for System $\mathrm{F_s}$ E-variables to create namespaces. It is difficult to have a symmetric expansion in System $\mathrm{F_s}$, because subtyping does not operate uniformly on typings (it is usually contravariant on the environment and covariant on the result type). It is possible to design System $\mathrm{F_s}$ with two kinds of E-variables (one, symmetric, to handle substitutions and $\forall$-introduction, and one, asymmetric, for subtyping), but it would make the system much more complex for no clear profit.

7.2 Type Inference in System F

Type inference in System F is undecidable [25]; however many different approaches have been conducted to circumvent this issue, by stratifying System F using a notion of rank, or by using type annotations to constrain type inference possibilities.

Giannini and Ronchi's type constraints. In [6], Giannini and Ronchi Della Rocca consider a syntax-directed version of System F. The authors define a notion of *typing scheme* σ, with a syntax similar to the one of System F types, except that quantifiers $\forall u.T$ contain placeholders u (called *sequence variables*), that can be replaced by a (possibly empty) set of type variables to give a System F type. For each term M, they also define a *principal typing scheme* $\Pi(M) = \langle D, \sigma, G, F\rangle$, where D is an environment that maps term variables to typing schemes, and G and F are constraints on the typing schemes occurring in σ or D that need to be satisfied. The set G contains subtyping constraints $\sigma_1 \leq_{\mathrm{F}} \sigma_2$, and F prevents certain quantifications from happening by restricting the possible values for the sequence variables u.

The principal typing scheme $\Pi(M)$ is similar to our initial skeletons; if $\Pi(M) = \langle D, \sigma, G, F\rangle$ and $Q \triangleright M : \langle A \vdash T\rangle/\Delta$ (with Q an initial skeleton for M), then D corresponds to A, σ to T, G to Δ, and F acts as the sets B that appear in E-variables $s^B\,T$. Any System F typing $\langle A \vdash T\rangle$ of M can be obtained from D, σ by applying a substitution (from type variables to types and sequence variables to set of type variables) which satisfies constraints G and F. This result corresponds to Theorem 5.4 in our system.

System $\mathrm{F_s}$ and the system of [6] differ mainly in their implementation. In particular, we have a mechanism to postpone subtyping (i.e., $\forall$-elimination), which does not have an equivalent in the system of Giannini and Ronchi. It seems that they do not need such mechanism, but to compensate for it, they have to generate more constraints when building their principal typing scheme $\Pi(M)$. We also believe that our system is easier to understand and easier to extend with other type constructors. Finally, Giannini and Ronchi define a notion of

rank over system F types (distinct from Leivant's rank based on the presence of polymorphism on the left of function types [13]), and provide for all n an inference algorithm for each restriction of their system to types of rank lower than n. We conjecture that this algorithm can be adapted to System F_s.

ML^F *and its variants.* ML^F [10, 11] is a conservative extension of ML at least as expressive as System F with principal types, i.e., result types whose instances (w.r.t. the ML^F type instance relation $\prec$) are exactly all possible result types for a term. The type system also enjoys decidable type inference (with a simple criterion on where type annotations are needed), and stability w.r.t. some program transformations, such as for example β-reduction and η-expansion.

ML^F types contain *flexible quantifiers* $\forall(a \succ \sigma)\sigma'$, which roughly represent sets of System F types of the form $[a := T]T'$, where T and T' are instances of the *type schemes* σ, σ'. For example, $\forall(a \succ \forall b(b \to b))(a \to a)$ represents the set $\{T \to T \mid \forall b(b \to b) \prec T\}$. With flexible quantifiers, terms that do not have a principal type in System F (w.r.t. the System F type instance relation) have a principal type in ML^F. Decidable type inference is obtained in ML^F by requiring type annotations on function parameters that are used two or more times with different type instances, so that the type inference algorithm never has to guess true polymorphism. *Rigid bindings* are used in ML^F types and typing rules to distinguish between inferred and annotated types. They are not necessary for decidable type inference, and can be removed at the cost of additional type annotations, as in HML [12].

Boxed polymorphism. Boxed polymorphism [9, 18] hides polymorphic types into boxes, considered as regular simple types. Several type systems follow this principle, such as PolyML [5], boxy types [24], and FPH [23]. We discuss only the most recent system, FPH. FPH is a type system based on System F, where boxes are used to mark where $\forall$-quantifiers have to be instantiated with polymorphic types. Provided that type annotations are given at these boxed positions, FPH type inference computes System F types (without any box) for terms. The system aims for simplicity for the programmer: only System F types are exposed, and writing type annotations does not require to think in term of boxes. Roughly, type annotations are necessary for λ-abstractions and let-bindings with *rich* types (i.e., types with quantifiers under arrow types). However, FPH is more restrictive than ML^F; more annotations are needed in general, and FPH terms admit principal types only for "box-free" types, not in general.

ML^F, FPH, and System F_s all aim for a modular type inference for System F types. It is difficult to compare our work to these two systems, because we do not propose a type inference algorithm for System F_s yet. In particular, assuming we follow their approach, we do not know how many annotations would be necessary to make System F_s type inference decidable. However, we can make the following observations. First, ML^F and FPH only infer result types, while our objective is to also infer complete typing, in order to have a fully compositional type inference algorithm. ML^F has principal types (w.r.t. to their instance relation),

while System $\mathrm{F_s}$ have initial skeletons, and FPH has principal types only for box-free types (where $\forall$-quantified variables cannot be instantiated with polymorphic types). $\mathsf{ML^F}$ types more terms than System F, while FPH and System $\mathrm{F_s}$ type the same terms as System F. Finally, FPH and System $\mathrm{F_s}$ are direct extensions of System F, and the constructions specific to these systems (the boxes and E-variables) can be kept away from the programmer most of the time (except in type error reports). On the other hand, $\mathsf{ML^F}$ types and type instance relation $\prec$ can be hard to understand, even in its simpler version HML.

To illustrate the differences between the three type systems, we consider the following example (taken from [11, 23]). Let $A = \mathsf{choose} : \forall a.(a \to a \to a), \mathsf{id} : \forall a.(a \to a)$ and $M = \mathsf{choose}\,@\,\mathsf{id}$. We can derive the following typing judgement for M:

$$\begin{aligned}
\mathrm{F_s} &: \langle A \vdash s_2^{\emptyset}\,((s_1^{\emptyset}\,\forall a.(a \to a)) \to (s_1^{\emptyset}\,\forall a.(a \to a)))\rangle \\
\mathsf{ML^F} &: \langle A \vdash \forall(a \succ \forall b(b \to b))(a \to a)\rangle \\
\mathrm{FPH} &: \langle A \vdash \forall b((b \to b) \to (b \to b))\rangle \\
&\quad \langle A \vdash \boxed{\forall b(b \to b)} \to \boxed{\forall b(b \to b)}\rangle
\end{aligned}$$

FPH can infer two result types for M, depending on the presence or absence of type annotations. These two incomparable types can be obtained from the (principal) $\mathsf{ML^F}$ type (ignoring the boxes), and also from the System $\mathrm{F_s}$ type, by applying the substitution $(s_2 := \forall b.\diamondsuit, s_1 := \diamondsuit^{:b\to b}, \boxdot)$ for the first one, and by simply erasing the E-variables for the second one.

Both System $\mathrm{F_s}$ E-variables and $\mathsf{ML^F}$ flexible bindings factor several System F types and typing derivations that are incomparable in System F, as shown with the $\mathsf{choose}\,@\,\mathsf{id}$ example. However, flexible bindings are more expressive and allow to type terms that are not typable in System F. Consider the example (taken from [11]) $\mathsf{let}\ x = (\mathsf{choose}\,@\,\mathsf{id})\ \mathsf{in}\ \mathsf{let}\ z = x\,@\,f\ \mathsf{in}\ x\,@\,g$, where $f : \forall a.(a \to a) \to \forall a.(a \to a), g : (b \to b) \to (b \to b)$. The $\mathsf{ML^F}$ type for $\mathsf{choose}\,@\,\mathsf{id}$ given above can be instantiated into the incomparable types of f and g. The term cannot be typed in System F nor in System $\mathrm{F_s}$. Adding quantification over E-variables would allow System $\mathrm{F_s}$ to type this term; we could type $\mathsf{choose}\,@\,\mathsf{id}$ with $\forall s.((s^{\emptyset}\,\forall a.(a \to a)) \to (s^{\emptyset}\,\forall a.(a \to a)))$ and instantiate s with different expansions to obtain the types of f and g. Adding quantification over E-variables should not raise any issue; we conjecture that it would allow System $\mathrm{F_s}$ to type as many terms as $\mathsf{ML^F}$. It would be interesting to see if there exists an encoding of $\mathsf{ML^F}$ types into System $\mathrm{F_s}$ types extended with quantified E-variables, and conversely. We leave this topic to future work.

8 Conclusion and Future Work

System $\mathrm{F_s}$ is an extension of System F with expansion, an operation originally defined in systems with intersection types. Expansion allows postponing the introduction of $\forall$-quantifiers and subtyping uses at an arbitrary nested position in a typing derivation. For any term M, we can generate an initial skeleton, from which we can obtain any System $\mathrm{F_s}$ judgement for M. We now give some ideas of follow-up on this work.

Type inference algorithm. To obtain decidable type inference in System $\mathrm{F_s}$, a first possibility is to use type annotations, as in ML^F or FPH. The question is then to know how many annotations are necessary compared to these two systems. Another idea is to study the link between constraints solving and semi-unification. Given a constraint $T_1 \leq T_2$, the semi-unification problem consists in finding S_1, S_2 so that $[S_2][S_1]T_1 = [S_1]T_2$. Vasconcellos et al. [22] used semi-unification to design and implement a type inference semi-algorithm for polymorphic recursion in Haskell. The authors claim that the algorithm terminates most of the time in practice. Maybe similar results can be obtained for System $\mathrm{F_s}$ as well. As discussed at the end of Section 5, System $\mathrm{F_s}$ can also be used to look for a subsystem of System F allowing for compositional type inference.

Mixing $\forall$-quantifiers and intersection types. A long-term goal is combining System E and System F into one system (called System EF), with both $\forall$-quantifiers and intersection types. With such a system, one could type a term with only intersection types, only System F types, or any combination of the two constructs, depending on the user's needs. Previous systems featuring both constructs (e.g. [15, 21]) do not use expansion variables; the main difficulty in mixing System E and System $\mathrm{F_s}$ is to make precise the interactions between the symmetric and asymmetric expansions. Maybe it is possible to define a more general expansion mechanism which supersedes the existing ones, and combine the two kinds of expansion variables into a single construct. A goal would be for System EF to have principal typings.

Because System E types all strongly normalizing terms, $\forall$-quantified types would only be used when required by the user when performing type inference in System EF. To this end, we could imagine various kinds of type annotations to mark positions within terms where System F types are required. These annotations could be complete types, such as $\lambda x^{\forall a.(a \to a)}.M$, or just type templates, such as $\lambda x^{(\forall a.*) \to *}.M$, meaning that the inferred type for x should be an arrow type, and the type of the argument should be a System F type. One could imagine different kinds of annotations at various positions in the term; we would like to see under which conditions (on both the annotations language and the positions in the term) the inference for such a system becomes decidable. The inference algorithm would then use intersection types by default, except for the marked positions where $\forall$-quantified types are requested.

References

1. S. Carlier, J. Polakow, J. B. Wells, and A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *ESOP*, volume 2986 of *LNCS*, pages 294–309. Springer, 2004.
2. S. Carlier and J. B. Wells. Expansion: the crucial mechanism for type inference with intersection types: A survey and explanation. *Electr. Notes Theor. Comput. Sci.*, 136:173–202, 2005.
3. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry:*

Essays on Combinatory Logic, Lambda Calculus, and Formalism, pages 535–560. Academic Press, 1980.
4. L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.
5. J. Garrigue and D. Rémy. Semi-explicit first-class polymorphism for ML. *Inf. Comput.*, 155(1-2):134–169, 1999.
6. P. Giannini and S. Ronchi Della Rocca. Type inference in polymorphic type discipline. In *TACS*, volume 526 of *LNCS*, pages 18–37. Springer, 1991.
7. J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.
8. A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theor. Comput. Sci.*, 311(1-3):1–70, 2004.
9. K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, 1994.
10. D. Le Botlan and D. Rémy. MLF: raising ML to the power of System F. In *ICFP*, pages 27–38. ACM, 2003.
11. D. Le Botlan and D. Rémy. Recasting MLF. *Inf. Comput.*, 207(6):726–785, 2009.
12. D. Leijen. Flexible types: robust type inference for first-class polymorphism. In *POPL*, pages 66–77. ACM, 2009.
13. D. Leivant. Polymorphic type inference. In *POPL*, pages 88–98, 1983.
14. S. Lenglet and J. B. Wells. Expansion for universal quantifiers. avalaible at `http://arxiv.org/abs/1201.1101`, 2012.
15. I. Margaria and M. Zacchi. Principal typing in a forall-and-discipline. *J. Log. Comput.*, 5(3):367–381, 1995.
16. R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
17. J. C. Mitchell. Polymorphic type inference and containment. *Inf. Comput.*, 76(2/3):211–249, 1988.
18. D. Rémy. Programming objects with ML-ART, an extension to ML with abstract and record types. In *TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346. Springer, 1994.
19. J. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 14 of *LNCS*, pages 408–423, London, UK, 1974. Springer-Verlag.
20. S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.
21. S. van Bakel, F. Barbanera, and M. Fernández. Polymorphic intersection type assignment for rewrite systems with abstractions and *beta*-rule. In *TYPES*, volume 1956 of *LNCS*, pages 41–60. Springer, 1999.
22. C. Vasconcellos, L. Figueiredo, and C. Camarão. Practical type inference for polymorphic recursion: an implementation in haskell. *J. UCS*, 9(8):873–890, 2003.
23. D. Vytiniotis, S. Weirich, and S. L. P. Jones. FPH: first-class polymorphism for Haskell. In *ICFP*, pages 295–306. ACM, 2008.
24. D. Vytiniotis, S. Weirich, and S. L. Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. In *ICFP*, pages 251–262. ACM, 2006.
25. J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1-3):111–156, 1999.
26. J. B. Wells. The essence of principal typings. In *ICALP*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.
27. J. B. Wells and C. Haack. Branching types. In *ESOP*, volume 2305 of *LNCS*, pages 115–132. Springer, 2002.