2-4-2 / Type systems
Polymorphism

François Pottier

December 2, 2008

INRIA

Contents

What is polymorphism?

Polymorphism is the ability for a term to *simultaneously* admit several distinct types.

Why polymorphism?

Polymorphism is *indispensable* [Reynolds, 1974]: if a function that sorts a list is independent of the type of the list elements, then it should be directly applicable to lists of integers, lists of Booleans, etc.

In short, it should have polymorphic type:

$$\forall X.(X \longrightarrow X \longrightarrow \text{bool}) \longrightarrow \text{list } X \longrightarrow \text{list } X$$

which *instantiates* to the monomorphic types:

$$(\text{int} \longrightarrow \text{int} \longrightarrow \text{bool}) \longrightarrow \text{list int} \longrightarrow \text{list int}$$
$$(\text{bool} \longrightarrow \text{bool} \longrightarrow \text{bool}) \longrightarrow \text{list bool} \longrightarrow \text{list bool}$$
$$\ldots$$

Why polymorphism?

In the absence of polymorphism, the only ways of achieving this effect would be:

- to manually duplicate the list sorting function at every type (*no-no!*);
- to use subtyping and claim that the function sorts lists of values of *any* type:

$$(\top \to \top \to \text{bool}) \to \text{list } \top \to \text{list } \top$$

(The type $\top$ is the type of all values, and the supertype of all types.) This leads to *loss of information* and subsequently requires introducing an unsafe *downcast* operation. This was the approach followed in Java before generics were introduced in 1.5.

Polymorphism is already implicitly present in simply-typed λ-calculus. Indeed, we have checked (in fact, inferred) that the type:

$$(X_1 \rightarrow X_2) \rightarrow X_1 \rightarrow X_1 \rightarrow X_2 \times X_2$$

is a *principal type* for the term $\lambda fxy.(f\ x, f\ y)$.

By saying that this term admits the polymorphic type:

$$\forall X_1 X_2.(X_1 \rightarrow X_2) \rightarrow X_1 \rightarrow X_1 \rightarrow X_2 \times X_2$$

we make polymorphism *internal* to the type system.

Polymorphism is a step on the road towards *type abstraction.*

Intuitively, if a function that sorts a list has polymorphic type:

$$\forall X.(X \to X \to \mathsf{bool}) \to \mathsf{list}\ X \to \mathsf{list}\ X$$

then it *knows nothing* about X — it is *parametric* in X — so it must manipulate the list elements *abstractly:* it can copy them around, pass them as arguments to the comparison function, but it cannot directly inspect their structure.

In short, within the code of the list sorting function, the variable X is an *abstract type*.

In the presence of polymorphism (and in the absence of effects), a type can reveal a lot of information about the terms that inhabit it. For instance, the polymorphic type

$$\forall X.X \rightarrow X$$

has only *one* inhabitant, namely the identity. Similarly, the type of the list sorting function reveals a *"free theorem"* about its behavior!

This phenomenon was studied by Reynolds [1983] and by Wadler [1989, 2007], among others. An account based on an operational semantics is offered by Pitts [2000].

Ad hoc versus parametric

Let me begin a short digression.

The term “polymorphism” dates back to a 1967 paper by Strachey [2000], where *ad hoc polymorphism* and *parametric polymorphism* were distinguished.

I see two different (and sometimes incompatible) ways of defining this distinction...

Ad hoc versus parametric: first definition

Here is one definition of the distinction:

With parametric polymorphism, a term can admit several types, all of which are *instances* of a single polymorphic type:

$$\text{int} \longrightarrow \text{int}, \mathit{bool} \longrightarrow \mathit{bool}, \ldots$$
$$\forall X.X \longrightarrow X$$

With ad hoc polymorphism, a term can admit a collection of *unrelated* types:

$$\text{int} \longrightarrow \text{int} \longrightarrow \text{int}, \text{float} \longrightarrow \text{float} \longrightarrow \text{float}, \ldots$$
$$\textit{but not } \forall X.X \longrightarrow X \longrightarrow X$$

Ad hoc versus parametric: second definition

Here is another definition:

With parametric polymorphism, *untyped programs have a well-defined semantics.* (Think of the identity function.) Types are used only to rule out unsafe programs.

With ad hoc polymorphism, untyped programs do not have a semantics: *the meaning of a term can depend upon its type.* (Think of an overloaded addition function.)

Ad hoc versus parametric: type classes

By the first definition, Haskell's *type classes* [Hudak *et al.*, 2007] are a form of (bounded) parametric polymorphism: terms have *principal (qualified) type schemes*, such as:

$$\forall X.\mathsf{Num}\ X \Rightarrow X \rightarrow X \rightarrow X$$

Yet, by the second definition, type classes are a form of ad hoc polymorphism: untyped programs do not have a semantics.

End of digression – in this course, we are interested only in the simplest form of parametric polymorphism.

Contents

The polymorphic λ-calculus (also known as: the second-order λ-calculus; F_2; System F) was independently defined by Girard (1972) and Reynolds [1974].

Compared to the simply-typed λ-calculus, types are extended:

$$T ::= \ldots \mid \forall X.T$$

How are terms are extended? There are two variants, which give rise to the *type-passing* and *type-erasing* interpretations of polymorphism...

Type-passing versus type-erasing interpretations

In the type-passing view, *types exist at runtime:* a value of type $\forall X.T$ is a function that expects a type as an argument.

In the type-erasing view, *types are erased prior to runtime:* a value of type $\forall X.T$ is a value that happens to simultaneously have type T for every X.

Type-passing polymorphic λ-calculus

In the type-passing variant [Reynolds, 1974], there are term-level constructs for introducing and eliminating the universal quantifier:

$$\text{TAbs}\ \frac{\Gamma; X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T} \qquad \text{TApp}\ \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t\ T' : [X \mapsto T']T}$$

Type variables are explicitly bound and appear in type environments.

The operational semantics is extended accordingly:

$$\begin{aligned} t &::= \dots \mid \Lambda X.t \mid t\ T \\ v &::= \dots \mid \Lambda X.t \\ E &::= \dots \mid []\ T \\ (\Lambda X.t)\ T &\longrightarrow [X \mapsto T]t \end{aligned}$$

Type-passing versus type-erasing: pros and cons

The type-passing interpretation has a number of disadvantages.

- because it alters the semantics, it does not fit our view that *the untyped semantics should pre-exist* and that a type system is only a predicate that selects a subset of the well-behaved terms.
- because it requires both values and types to exist at runtime, it can lead to a *duplication of machinery*. Compare type-preserving closure conversion in type-passing [Minamide et al., 1996] and in type-erasing [Morrisett et al., 1999] styles.

An apparent advantage of the type-passing interpretation is to allow *typecase*; however, typecase can be simulated in a type-erasing system by viewing runtime *type descriptions* as *values* [Crary et al., 2002].

In the following, we focus on the *type-erasing* variant, which does not alter the syntax or semantics of untyped terms.

The syntax and semantics of terms are unchanged.

The typing rules that introduce and eliminate the universal quantifier are non-syntax-directed:

∀-Intro
$$\frac{\Gamma \vdash t : T \qquad X \# \Gamma}{\Gamma \vdash t : \forall X.T}$$

∀-Elim
$$\frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t : [X \mapsto T']T}$$

Because this type-erasing variant of System F allows evaluation under a universal introduction rule, it exhibits an interaction with references, while the type-passing variant does not. (Details later today.)

Type variables are not explicitly introduced.

Why the side condition $X \# \Gamma$?...

Omitting the side condition leads to *unsoundness*:

$$\dfrac{x : X_1 \vdash x : X_1}{\rule{0pt}{0pt}}$$

Omitting the side condition leads to *unsoundness:*

$$\text{Broken } \forall\text{-Intro}\ \dfrac{x : X_1 \vdash x : X_1}{x : X_1 \vdash x : \forall X_1.X_1}$$

Omitting the side condition leads to *unsoundness:*

$$\dfrac{\dfrac{\dfrac{x : X_1 \vdash x : X_1}{x : X_1 \vdash x : \forall X_1.X_1}\ \text{Broken } \forall\text{-Intro}}{x : X_1 \vdash x : X_2}\ \forall\text{-Elim}}{\quad}$$

Omitting the side condition leads to *unsoundness:*

$$\cfrac{\cfrac{\cfrac{\cfrac{x : X_1 \vdash x : X_1}{x : X_1 \vdash x : \forall X_1.X_1}\text{Broken }\forall\text{-Intro}}{x : X_1 \vdash x : X_2}\forall\text{-Elim}}{\emptyset \vdash \lambda x.x : X_1 \to X_2}\text{Abs}}{}$$

Omitting the side condition leads to *unsoundness:*

$$
\dfrac{\dfrac{\dfrac{\dfrac{x : X_1 \vdash x : X_1}{x : X_1 \vdash x : \forall X_1.X_1}\text{Broken }\forall\text{-Intro}}{x : X_1 \vdash x : X_2}\forall\text{-Elim}}{\emptyset \vdash \lambda x.x : X_1 \rightarrow X_2}\text{Abs}}{\emptyset \vdash \lambda x.x : \forall X_1.\forall X_2.X_1 \rightarrow X_2}\forall\text{-Intro}^2
$$

This is a type derivation for a *type cast* (Objective Caml's Obj.magic).

A good intuition is: a judgement $\Gamma \vdash t : T$ corresponds to the logical assertion $\forall \bar{X}.(\Gamma \Rightarrow T)$, where $\bar{X}$ are the free type variables of the judgement.

In that view, $\forall$-Intro corresponds to the axiom:

$$\forall X.(P \Rightarrow Q) \quad \equiv \quad P \Rightarrow (\forall X.Q) \qquad \text{if } X \# P$$

On the side condition $X \# \Gamma$

Quiz: why is there no such side condition in the type-passing variant of System F? Or is there one, and where? back

Quiz: why is there no such side condition in the type-passing variant of System F? Or is there one, and where? back

Answer: no such condition is needed in rule TAbs, because (1) in the premise of TAbs, the environment is extended with an explicit binding of X, and (2) the definition of environment lookup, not shown earlier, contains a side condition:

$$(\Gamma; X)(x) \quad = \quad \Gamma(x) \qquad \text{if } X \# \Gamma(x)$$

The details vary, but the side condition exists in both variants.

Here is a version of the term $\lambda fxy.(f\ x, f\ y)$ that carries explicit type abstractions and annotations:

$$\Lambda X_1.\Lambda X_2.\lambda f : X_1 \rightarrow X_2.\lambda x : X_1.\lambda y : X_1.(f\ x, f\ y)$$

This term admits the polymorphic type:

$$\forall X_1.\forall X_2.(X_1 \rightarrow X_2) \rightarrow X_1 \rightarrow X_1 \rightarrow X_2 \times X_2$$

Quite unsurprising, right?

Perhaps more surprising is the fact that this untyped term can be decorated in a different way:

$$\Lambda X_1.\Lambda X_2.\lambda f : \forall X.X \to X.\lambda x : X_1.\lambda y : X_2.(f\ X_1\ x, f\ X_2\ y)$$

This term admits the polymorphic type:

$$\forall X_1.\forall X_2.(\forall X.X \to X) \to X_1 \to X_2 \to X_1 \times X_2$$

This begs the question: ...

Which of the two is more general?

$$\forall X_1.\forall X_2.(X_1 \to X_2) \to X_1 \to X_1 \to X_2 \times X_2$$
$$\forall X_1.\forall X_2.(\forall X.X \to X) \to X_1 \to X_2 \to X_1 \times X_2$$

Which of the two is more general?

$$\forall X_1.\forall X_2.(X_1 \rightarrow X_2) \rightarrow X_1 \rightarrow X_1 \rightarrow X_2 \times X_2$$
$$\forall X_1.\forall X_2.(\forall X.X \rightarrow X) \rightarrow X_1 \rightarrow X_2 \rightarrow X_1 \times X_2$$

One requires x and y to admit a common type, while the other requires f to be polymorphic.

Neither of these types can be an instance of the other, for any reasonable definition of the word "instance", because *each* has an inhabitant that *does* not admit the other as a type.

(Exercise: find these inhabitants!)

It seems plausible that the untyped term $\lambda fxy.(f\ x, f\ y)$ does not admit a type of which both of these types are instances.

But, in order to prove this, one would have to fix what it means for T_2 to be an *instance* of T_1 — or, equivalently, for T_1 to be *more general* than T_2.

Several definitions are possible...

In System *F*, "to be an instance" is usually *defined* by the rule:

$$\text{InstGen}\quad \frac{\bar{Y} \# \forall \bar{X}.T}{\forall \bar{X}.T \leq \forall \bar{Y}.[\vec{X} \mapsto \vec{T}]T}$$

One can show that, if $T_1 \leq T_2$, then any term that has type T_1 also has type T_2; that is, the following rule is *admissible:*

$$\text{Sub}\quad \frac{\Gamma \vdash t : T_1 \qquad T_1 \leq T_2}{\Gamma \vdash t : T_2}$$

(Not-so-easy *exercise:* prove it!)

Another syntactic notion of instance: System F_η

Mitchell [1988] defines System F_η, a version of System F extended with a richer *instance* relation:

InstGen

$$\frac{\bar{Y} \# \forall\bar{X}.T}{\forall\bar{X}.T \leq \forall\bar{Y}.[\vec{X} \mapsto \vec{T}]T}$$

Distributivity

$$\forall\bar{X}.(T_1 \to T_2) \leq (\forall\bar{X}.T_1) \to (\forall\bar{X}.T_2)$$

Congruence-$\to$

$$\frac{T_2 \leq T_1 \qquad T_1' \leq T_2'}{T_1 \to T_1' \leq T_2 \to T_2'}$$

Congruence-$\forall$

$$\frac{T_1 \leq T_2}{\forall X.T_1 \leq \forall X.T_2}$$

Transitivity

$$\frac{T_1 \leq T_2 \qquad T_2 \leq T_3}{T_1 \leq T_3}$$

In System F_η, Sub is an explicit rule.

System F_η can also be defined as the closure of System F under η-equality.

Why is a rich notion of instance potentially interesting?

A definition of principal typings

Ideally, a type system should satisfy the *principal typings* property [Wells, 2002]:

> *Every well-typed term t admits a principal typing — one that is more general than every other typing of t.*

Whether this property holds depends on a definition of *instance*. The more liberal the instance relation, the more hope there is of having principal typings.

A "semantic" notion of instance

Wells [2002] notes that, *once a type system is fixed*, a most liberal notion of instance can be defined, *a posteriori*, by:

> *A typing θ_1 is more general than a typing θ_2 if and only if every term that admits θ_1 admits θ_2 as well.*

This is the largest reasonable notion of instance: $\leq$ is *defined* as the largest relation such that a subtyping principle is admissible.

This definition can be used to prove that a system does *not* have principal typings, under *any* reasonable definition of "instance".

Which systems have principal typings?

We have seen that *simply-typed λ-calculus has principal typings,* with respect to a substitution-based notion of instance.

Wells [2002] shows that *neither System F nor System F_η have principal typings.*

It was shown earlier that *System F_η's instance relation is undecidable* [Wells, 1995, Tiuryn and Urzyczyn, 2002] and that *type inference for both System F and System F_η is undecidable* [Wells, 1999].

Which systems have principal typings?

There are still a few positive results...

Some systems of *intersection types* have principal typings [Wells, 2002] — but they are very complex.

Damas and Milner's type system (coming up next) has *principal types* and *decidable type inference*.

Damas and Milner's type system

Damas and Milner's type system [Milner, 1978] offers a restricted form of polymorphism, while avoiding the difficulties associated with type inference in System *F*.

This type system is at the heart of Standard ML, Objective Caml, and Haskell.

The type inference algorithm should be a simple extension of the algorithm that was developed for simply-typed λ-calculus.

To this end, it should exploit polymorphism where obviously *available*, but should not try to guess where polymorphism is *necessary*.

In other words, it should continue to rely on first-order unification: that is, type variables should continue to stand for *types without quantifiers*.

For instance, this term should be well-typed:

$$\text{let } f = \lambda z.z \text{ in } (f\ 0, f\ \text{true})$$

Indeed, f is *known* to be bound to $\lambda z.z$, a term whose principal type ($\forall X.X \rightarrow X$) can be inferred as in simply-typed λ-calculus.

On the other hand, this term should be ill-typed:

$$\lambda f.(f\ 0, f\ \text{true})$$

Indeed, no monotype is suitable for f, and we deliberately refuse to let a type variable stand for an unknown polymorphic type.

In short, *let-bound* variables receive possibly polymorphic types, while *λ-bound* variables must receive monomorphic types.

There is a simple intuition behind Damas and Milner's type system: a closed term has type T if and only if its *let-normal form* has type T in simply-typed λ-calculus.

A term's let-normal form is obtained by iterating the rewrite rule:

$$\text{let } x = t_1 \text{ in } t_2 \quad \longrightarrow \quad t_1; [x \mapsto t_1]t_2$$

This intuition suggests type-checking and type inference algorithms. But these algorithms are *not practical,* because:

- they have exponential complexity;
- separate compilation blocks reduction to let-normal form.

In the following, we study a direct presentation of Damas and Milner's type system, which *does* not involve let-normal forms.

It is *practical*, because:

- it leads to an efficient type inference algorithm;
- it supports separate compilation.

Terms are now given by:

$$t ::= x \mid \lambda x.t \mid t\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \mid \ldots$$

The *let* construct is no longer sugar for a β-redex: it is now a primitive form.

The syntax of *types* is unchanged with respect to simply-typed λ-calculus:

$$T ::= X \mid T \to T \mid \ldots$$

A separate category of *type schemes* is introduced:

$$S ::= \forall \bar{X}.T$$

These correspond to the principal type schemes of simply-typed λ-calculus. All quantifiers must appear in *prenex position*, so type schemes are less expressive than System F types.

A type environment Γ is now a finite sequence of bindings of variables to *type schemes*.

Judgements now take the form:

$$\Gamma \vdash t : S$$

Types form a subset of type schemes, so type environments and judgements can contain types too.

Here is a standard, non-syntax-directed presentation.

Var
$$\frac{\Gamma(x) = S}{\Gamma \vdash x : S}$$

Abs
$$\frac{\Gamma; x : T \vdash t : T'}{\Gamma \vdash \lambda x.t : T \to T'}$$

App
$$\frac{\Gamma \vdash t_1 : T \to T' \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ t_2 : T'}$$

Let
$$\frac{\Gamma \vdash t_1 : S \qquad \Gamma; x : S \vdash t_2 : T}{\Gamma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : T}$$

Gen
$$\frac{\Gamma \vdash t : T \qquad \bar{X} \# \Gamma}{\Gamma \vdash t : \forall \bar{X}.T}$$

Inst
$$\frac{\Gamma \vdash t : \forall \bar{X}.T}{\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T}$$

Let moves a type scheme into the environment, which Var can exploit.

Abs and App are unchanged. *λ-bound variables receive a monotype.*

Gen and Inst are as in type-erasing System F, except they introduce or eliminate multiple universal quantifiers at once. *Type variables are instantiated with monotypes.*

Example

Here is a simple type derivation that exploits polymorphism:

$$
\dfrac{
\text{Gen}\;\dfrac{\text{Abs}\;\dfrac{\text{Var}\;\dfrac{}{z : X \vdash z : X}}{\emptyset \vdash \lambda z.z : X \to X}}{\emptyset \vdash \lambda z.z : \color{blue}{\forall X.X \to X}}
\qquad
\dfrac{
\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \color{blue}{\forall X.X \to X}}\;\text{Var}}{\Gamma \vdash f : \mathsf{int} \to \mathsf{int}}\;\text{Inst}}{\Gamma \vdash f\; 0 : \mathsf{int}}\;\text{App}
\quad
\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \color{blue}{\forall X.X \to X}}\;\text{Var}}{\Gamma \vdash f : \mathsf{bool} \to \mathsf{bool}}\;\text{Inst}}{\Gamma \vdash f\; \mathsf{true} : \mathsf{bool}}\;\text{App}
}{\Gamma \vdash (f\; 0, f\; \mathsf{true}) : \mathsf{int} \times \mathsf{bool}}\;\text{Pair}
}{\emptyset \vdash \mathsf{let}\; f = \lambda z.z \;\mathsf{in}\; (f\; 0, f\; \mathsf{true}) : \mathsf{int} \times \mathsf{bool}}\;\text{Let}
$$

(Γ stands for $f : \forall X.X \to X$.)

Gen is used above Let (at left), and Inst is used below Var. In fact, every type derivation can be put in this form. forward

A non-example

As announced, this term is *ill-typed:*

$$\lambda f.(f\ 0, f\ \mathsf{true})$$

A non-example

As announced, this term is *ill-typed:*

$$\lambda f.(f\ 0, f\ \mathsf{true})$$

Indeed, this term contains no “let” construct, so it is type-checked exactly as in simply-typed λ-calculus, where it is ill-typed, because the equation:

$$\mathsf{int} \rightarrow T_1 = \mathsf{bool} \rightarrow T_2$$

has no solution.

Recall that this term is well-typed in type-erasing System F.

Contents

Outline

Type soundness for Damas and Milner's type system is proved using the standard syntactic method [Wright and Felleisen, 1994].

Before reviewing the Subject Reduction proof, we need two stepping stones:

- a *Type Substitution* lemma;
- a *syntax-directed* presentation of the type system.

Definition

A *renaming* ρ is a total, bijective mapping of type variables to type variables whose domain is finite. The *domain* of ρ is the set of the type variables X such that $\rho(X) \neq X$. The *support* of ρ is its domain.

Renamings apply to types, type schemes, and type environments:

$$\begin{aligned} \rho(T_1 \rightarrow T_2) &= \rho(T_1) \rightarrow \rho(T_2) \\ \rho(\forall \bar{X}.T) &= \forall \rho(\bar{X}).\rho(T) \\ \rho(\emptyset) &= \emptyset \\ \rho(\Gamma; x : S) &= \rho(\Gamma); x : \rho(S) \end{aligned}$$

The *skeleton* of a type derivation is its underlying rule name tree. Two derivations are *isomorphic* when they have the same skeleton.

Lemma (Renaming)

For every derivation of $\Gamma \vdash t : S$*, there exists an isomorphic derivation of* $\rho(\Gamma) \vdash t : \rho(S)$.

Proof.

No typing rule is sensitive to the choice of type variable names. □

Definition

A *substitution* φ is a total mapping of type variables to types whose domain is finite. The *domain* of φ is the set of the type variables X such that $\varphi(X) \neq X$. The *codomain* of φ is the set of the type variables that appear free in the image of its domain. The *support* of φ is the union of its domain and codomain. $X \# \varphi$ holds if and only if X is not in the support of φ.

Substitutions apply to types, type schemes, and type environments:

$$
\begin{aligned}
\varphi(T_1 \rightarrow T_2) &= \varphi(T_1) \rightarrow \varphi(T_2) \\
\varphi(\forall \bar{X}.T) &= \forall \bar{X}.\varphi(T) \qquad \text{if } \bar{X} \# \varphi \\
\varphi(\emptyset) &= \emptyset \\
\varphi(\Gamma; x : S) &= \varphi(\Gamma); x : \varphi(S)
\end{aligned}
$$

Lemma (Type substitution)

For every derivation of $\Gamma \vdash t : S$*, there exists an isomorphic derivation of* $\varphi(\Gamma) \vdash t : \varphi(S)$*.*

Proof.

By structural induction over derivations. Only two cases are of interest, namely Gen and Inst. See next slides... □

The hypothesis is:

$$\frac{\Gamma \vdash t : T \qquad \bar{X} \# \Gamma}{\Gamma \vdash t : \forall \bar{X}.T}$$

The goal is:

$$\varphi(\Gamma) \vdash t : \varphi(\forall \bar{X}.T)$$

How to proceed? (Hint: what do we know about $\varphi(\forall \bar{X}.T)$?)

We distinguish two cases:

- first, the ideal case where $\bar{X} \# \varphi$ holds; there, the goal becomes:

$$\varphi(\Gamma) \vdash t : \forall \bar{X}.\varphi(T)$$

- then, the general case.

Invoking the induction hypothesis yields $\varphi(\Gamma) \vdash t : \varphi(T)$.

Invoking the induction hypothesis yields $\varphi(\Gamma) \vdash t : \varphi(T)$.

The freshness hypothesis $\bar{X} \# \varphi$ and the premise $\bar{X} \# \Gamma$, imply $\bar{X} \# \varphi(\Gamma)$ (lemma — *exercise!*).

Invoking the induction hypothesis yields $\varphi(\Gamma) \vdash t : \varphi(T)$.

The freshness hypothesis $\bar{X} \# \varphi$ and the premise $\bar{X} \# \Gamma$, imply $\bar{X} \# \varphi(\Gamma)$ (lemma — *exercise!*).

We now build a new instance of Gen:

$$\frac{\varphi(\Gamma) \vdash t : \varphi(T) \qquad \bar{X} \# \varphi(\Gamma)}{\varphi(\Gamma) \vdash t : \forall \bar{X}.\varphi(T)}$$

This is the goal.

What if $\bar{X} \# \varphi$ does not hold?

Recall that the hypothesis is:

$$\frac{\Gamma \vdash t : T \qquad \bar{X} \# \Gamma}{\Gamma \vdash t : \forall \bar{X}.T}$$

What if $\bar{X} \# \varphi$ does not hold?

Recall that the hypothesis is:

$$\frac{\Gamma \vdash t : T \qquad \bar{X} \# \Gamma}{\Gamma \vdash t : \forall \bar{X}.T}$$

This is where the premise $\bar{X} \# \Gamma$ plays a role.

Type substitution: Gen / general case

What if $\bar{X} \# \varphi$ does not hold?

Recall that the hypothesis is:

$$\frac{\Gamma \vdash t : T \qquad \bar{X} \# \Gamma}{\Gamma \vdash t : \forall \bar{X}.T}$$

This is where the premise $\bar{X} \# \Gamma$ plays a role.

Because *$\bar{X}$ does not appear free in the conclusion,* it can be renamed in the premises (via the renaming lemma) without affecting the conclusion.

We are then back to the ideal case, with a different choice of $\bar{X}$.

The hypothesis is:

$$\frac{\Gamma \vdash t : \forall \bar{X}.T}{\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T}$$

The goal is:

$$\varphi(\Gamma) \vdash t : \varphi([\vec{X} \mapsto \vec{T}]T)$$

How to proceed?

We again begin with the ideal case where $\bar{X} \# \varphi$ holds.

We again begin with the ideal case where $\bar{X} \# \varphi$ holds.

By the induction hypothesis, we have:

$$\varphi(\Gamma) \vdash t : \varphi(\forall \bar{X}.T)$$

Type substitution: Inst / ideal case

We again begin with the ideal case where $\bar{X} \# \varphi$ holds.

By the induction hypothesis, we have:

$$\varphi(\Gamma) \vdash t : \varphi(\forall \bar{X}.T)$$

which, by the freshness hypothesis, can be written:

$$\varphi(\Gamma) \vdash t : \forall \bar{X}.\varphi(T)$$

Type substitution: Inst / ideal case

We again begin with the ideal case where $\bar{X} \# \varphi$ holds.

By the induction hypothesis, we have:

$$\varphi(\Gamma) \vdash t : \varphi(\forall \bar{X}.T)$$

which, by the freshness hypothesis, can be written:

$$\varphi(\Gamma) \vdash t : \forall \bar{X}.\varphi(T)$$

We now build a new instance of Inst:

$$\frac{\varphi(\Gamma) \vdash t : \forall \bar{X}.\varphi(T)}{\varphi(\Gamma) \vdash t : [\bar{X} \mapsto \varphi(\vec{T})]\varphi(T)}$$

Is this the goal $\varphi(\Gamma) \vdash t : \varphi([\vec{X} \mapsto \vec{T}]T)$?

There remains to check that the substitutions $\varphi_1 = \varphi \circ [\vec{X} \mapsto \vec{T}]$ and $\varphi_2 = [\vec{X} \mapsto \varphi(\vec{T})] \circ \varphi$ coincide.

This is done by applying both substitutions to an arbitrary variable X.

We distinguish two sub-cases: $X \in \bar{X}$ and $X \notin \bar{X}$.

Recall $\varphi_1 = \varphi \circ [\vec{X} \mapsto \vec{T}]$ and $\varphi_2 = [\vec{X} \mapsto \varphi(\vec{T})] \circ \varphi$.

Type substitution: Inst / ideal case / sub-case $X \in \bar{X}$

Recall $\varphi_1 = \varphi \circ [\vec{X} \mapsto \vec{T}]$ and $\varphi_2 = [\vec{X} \mapsto \varphi(\vec{T})] \circ \varphi$.

For some index i, X is X_i, the i-th element of the vector $\vec{X}$. Then, $\varphi_1(X)$ is $\varphi(T_i)$, where T_i is the i-th element of the vector $\vec{T}$.

Type substitution: Inst / ideal case / sub-case $X \in \bar{X}$

Recall $\varphi_1 = \varphi \circ [\vec{X} \mapsto \vec{T}]$ and $\varphi_2 = [\vec{X} \mapsto \varphi(\vec{T})] \circ \varphi$.

For some index i, X is X_i, the i-th element of the vector $\vec{X}$. Then, $\varphi_1(X)$ is $\varphi(T_i)$, where T_i is the i-th element of the vector $\vec{T}$.

$X \in \bar{X}$ and $\bar{X} \# \varphi$ imply $X \# \varphi$, so X is not in the domain of φ, so $\varphi(X)$ is X. There follows that $\varphi_2(X)$ is also $\varphi(T_i)$.

Type substitution: Inst / ideal case / sub-case $X \notin \bar{X}$

Recall $\varphi_1 = \varphi \circ [\vec{X} \mapsto \vec{T}]$ and $\varphi_2 = [\vec{X} \mapsto \varphi(\vec{T})] \circ \varphi$.

Recall $\varphi_1 = \varphi \circ [\vec{X} \mapsto \vec{T}]$ and $\varphi_2 = [\vec{X} \mapsto \varphi(\vec{T})] \circ \varphi$.

Then, $\varphi_1(X)$ is $\varphi(X)$.

Recall $\varphi_1 = \varphi \circ [\vec{X} \mapsto \vec{T}]$ and $\varphi_2 = [\vec{X} \mapsto \varphi(\vec{T})] \circ \varphi$.

Then, $\varphi_1(X)$ is $\varphi(X)$.

$\bar{X} \# \varphi$ and $\bar{X} \# X$ imply $\bar{X} \# \varphi(X)$, which implies that $\varphi_2(X)$ is $\varphi(X)$.

Type substitution: Gen / general case

What if $\bar{X} \# \varphi$ does not hold?

Recall that the hypothesis is:

$$\frac{\Gamma \vdash t : \forall \bar{X}.T}{\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T}$$

What if $\bar{X} \# \varphi$ does not hold?

Recall that the hypothesis is:

$$\frac{\Gamma \vdash t : \forall \bar{X}.T}{\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T}$$

Because *$\bar{X}$ is mute in the premise* (where it is bound) *and in the conclusion* (where it is substituted out), it can be renamed without affecting either of them.

We are then back to the ideal case, with a different choice of $\bar{X}$.

What if you don't believe me!?

Isn't there too much handwaving in these alpha-conversion arguments?

True. It would be easy to get one of them wrong.

Confidence can be increased via *mechanized proof-checking*.

However, how to understand name binding, and how to deal with it in a logic or a proof assistant, is still partly an open issue.

For theoretical bases, see Gabbay and Pitts [2002] and Pitts [2006]. There is also work within proof assistants, *e.g.* Coq [Aydemir *et al.*, 2008, Chlipala, 2008], Isabelle/HOL [Urban and Tasson, 2005], Twelf [Harper and Licata, 2007, Pientka, 2007].

An instance of Gen followed with an instance of Inst annihilate.

Lemma (Annihilation)

If $\Gamma \vdash t : S$ admits a derivation with skeleton Δ/Gen/Inst, then it admits a derivation with skeleton Δ.

Proof.

By the Type Substitution lemma (see next slides)... □

Up to a renaming of Gen's premise, the hypothesis is:

$$\text{Inst}\ \dfrac{\text{Gen}\ \dfrac{\Gamma \vdash t : T \qquad \bar{X} \,\#\, \Gamma}{\Gamma \vdash t : \forall \bar{X}.T}}{\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T}$$

where the derivation of $\Gamma \vdash t : T$ has skeleton Δ.

Up to a renaming of Gen's premise, the hypothesis is:

$$\text{Inst}\ \dfrac{\text{Gen}\ \dfrac{\Gamma \vdash t : T \qquad \bar{X} \mathrel{\#} \Gamma}{\Gamma \vdash t : \forall \bar{X}.T}}{\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T}$$

where the derivation of $\Gamma \vdash t : T$ has skeleton Δ.

By the Type Substitution lemma, there is a derivation of:

$$[\vec{X} \mapsto \vec{T}]\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T$$

with skeleton Δ.

Up to a renaming of Gen's premise, the hypothesis is:

$$\text{Inst}\ \dfrac{\text{Gen}\ \dfrac{\Gamma \vdash t : T \qquad \bar{X} \# \Gamma}{\Gamma \vdash t : \forall \bar{X}.T}}{\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T}$$

where the derivation of $\Gamma \vdash t : T$ has skeleton Δ.

By the Type Substitution lemma, there is a derivation of:

$$[\vec{X} \mapsto \vec{T}]\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T$$

with skeleton Δ.

Because $\bar{X} \# \Gamma$, this is exactly:

$$\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T$$

In Damas and Milner's type system, back a non-trivial instance of Gen cannot appear above Abs, App, Let (at right), or Gen. It *can* appear above Let (at left) or Inst.

A non-trivial instance of Inst cannot appear below Abs, App, Let, or Inst. It *can* appear below Var or Gen.

The Annihilation lemma implies that disallowing Gen above Inst removes no expressive power.

In summary, *Gen is useful only above Let (at left)*, or possibly at the root of the derivation; and *Inst is useful only below Var.*

This leads to an alternative formulation of the type system...

Here is the standard, *syntax-directed* presentation of Damas and Milner's type system.

$$\textsc{VarInst}\quad \frac{\Gamma(x) = \forall\bar{X}.T}{\Gamma \vdash x : [\vec{X} \mapsto \vec{T}]T}$$

$$\textsc{Abs}\quad \frac{\Gamma; x : T \vdash t : T'}{\Gamma \vdash \lambda x.t : T \rightarrow T'}$$

$$\textsc{App}\quad \frac{\Gamma \vdash t_1 : T \rightarrow T' \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ t_2 : T'}$$

$$\textsc{GenLet}\quad \frac{\Gamma \vdash t_1 : T_1 \qquad \bar{X} \# \Gamma \qquad \Gamma; x : \forall\bar{X}.T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}$$

Judgements are now of the form $\Gamma \vdash t : T$.

The two presentations are equivalent:

Lemma (Equivalence)

Let $\bar{X} \# \Gamma$. The non-syntax-directed presentation derives $\Gamma \vdash t : \forall\bar{X}.T$ if and only if the syntax-directed presentation derives $\Gamma \vdash t : T$.

This is good to know in itself.

Furthermore, this means that, in the subject reduction proof that follows, we can *deconstruct syntax-directed derivations* (nice, because there are fewer) and *build non-syntax-directed derivations* (nice, because there are more).

As in the simply-typed λ-calculus, we prove a straightforward value substitution lemma:

Lemma (Value substitution)

$x : S, \Gamma \vdash t : T$ *and* $x \notin \mathrm{dom}(\Gamma)$ *and* $\emptyset \vdash v : S$ *imply* $\Gamma \vdash [x \mapsto v]t : T$.

Here, the lemma is formulated in terms of the original presentation of the system.

By the way, this means that, if a term is well-typed, then so is its let-normal form. The converse is also true, and will be shown when we study type inference.

To prove subject reduction, we assume that *the syntax-directed presentation* derives $\Gamma \vdash t : T$, we assume $t \longrightarrow t'$, and check that $\Gamma \vdash t' : T$ holds *in the original presentation*.

The proof is immediate:

- Case (β): *deconstruct* App and Abs, then apply the value substitution lemma;
- Case (let): *deconstruct* GenLet, then apply Gen and the value substitution lemma;
- Case (context): routine.

Progress is proved just as in the simply-typed λ-calculus, working on the syntax-directed presentation.

Bottom line

In summary, Type Substitution and Annihilation are the key properties that make the type system sound.

For further reading, see Wright and Felleisen [1994], Pierce [2002], Pottier and Rémy [2005].

Contents

In the last course (November 25, 2008), we noted that the program:

$$\text{let } x = \text{ref } 3 \text{ in } (x := 1; !x)$$

does not have the same semantics as its let-normal form:

$$\text{ref } 3; (\text{ref } 3) := 1; !(\text{ref } 3)$$

In the presence of effects, a term and its let-normal form do not have the same semantics, so the naïve approach to polymorphism, based on let-normal forms, back has *no reason to be sound.*

Damas and Milner's type system, which derives the same (monomorphic) typings as the naïve approach, has no reason to be sound either...

In the last course, we also noted that type soundness strongly relies on the fact that *every reference cell has a fixed type*.

So, it is important to *rule out polymorphic references:* cells that admit multiple types at once. In short, a type of the form:

$$\forall X.\mathsf{ref}\ T$$

(where X appears in T) should never be inhabited.

Right?

Right! Yet, if naïvely *extended* with references, Damas and Milner's type system allows constructing polymorphic references.

This well-typed program, where x receives the type scheme $\forall X.\mathsf{ref}\ (X \longrightarrow X)$, *goes wrong:*

$$\mathsf{let}\ x = \mathsf{ref}\ (\lambda z.z)\ \mathsf{in}\ x := (\lambda z.z + 1); !x\ \mathsf{true}$$

The cell x is written at type $\mathsf{int} \longrightarrow \mathsf{int}$, then read at type $\mathsf{bool} \longrightarrow \mathsf{bool}$.

We have proved type soundness for references without polymorphism, and for polymorphism without references, but *the combination fails.* Ah!

Let's review the proof for references and polymorphism together.

First, we augment typing judgements so that they take the form:

$$M, \Gamma \vdash t : S$$

where M is a store typing, which maps memory locations to...

First, we augment typing judgements so that they take the form:

$$M, \Gamma \vdash t : S$$

where M is a store typing, which maps memory locations to... *types.*

No choice here: the syntax of types is $T ::= \ldots \mid \text{ref}\, T$, not $T ::= \ldots \mid \text{ref}\, S$, so the contents of a cell must have monomorphic type.

This restriction is imposed by the design of ML. It is not required for soundness. In System F with references, a type of the form $\text{ref}\,(\forall X.T)$ would be fine.

The two novel rules of Damas and Milner's type system become:

$$\text{Gen}\quad \frac{M, \Gamma \vdash t : T \qquad \bar{X} \mathrel{\#} \Gamma}{M, \Gamma \vdash t : \forall \bar{X}.T} \qquad\qquad \text{Inst}\quad \frac{M, \Gamma \vdash t : \forall \bar{X}.T}{M, \Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T}$$

Right?

Clarifying the typing rules

The two novel rules of Damas and Milner's type system become:

Gen
$$\frac{M, \Gamma \vdash t : T \qquad \bar{X} \# \Gamma}{M, \Gamma \vdash t : \forall \bar{X}.T}$$

Inst
$$\frac{M, \Gamma \vdash t : \forall \bar{X}.T}{M, \Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T}$$

Right?

No way! This version of Gen is broken. Because $\bar{X}$ can appear in M, the Type Substitution lemma does not hold. So...

The correct rule is, of course:

Gen

$$\frac{M, \Gamma \vdash t : T \qquad \bar{X} \# M, \Gamma}{M, \Gamma \vdash t : \forall \bar{X}.T}$$

Mysterious slogan #1: *one must not generalize a type variable that appears in the store typing.* Aha!

This version satisfies Type Substitution.

Yet, the counter-example program shows that Subject Reduction is still broken... Where is the bug?

The problem lies in the (context) case of the Subject Reduction proof, and more specifically in the case of reduction under a universal introduction rule.

The hypotheses are:

$$\frac{M, \emptyset \vdash t : T \qquad \bar{X} \# M}{M, \emptyset \vdash t : \forall \bar{X}.T} \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

The problem lies in the (context) case of the Subject Reduction proof, and more specifically in the case of reduction under a universal introduction rule.

The hypotheses are:

$$\frac{M, \emptyset \vdash t : T \qquad \bar{X} \# M}{M, \emptyset \vdash t : \forall \bar{X}.T} \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

By the induction hypothesis, there exists M' such that:

$$M', \emptyset \vdash t' : T \quad \text{and} \quad \vdash \mu' : M' \quad \text{and} \quad M \subseteq M'$$

Nailing the bug

The problem lies in the (context) case of the Subject Reduction proof, and more specifically in the case of reduction under a universal introduction rule.

The hypotheses are:

$$\frac{M, \emptyset \vdash t : T \qquad \bar{X} \mathrel{\#} M}{M, \emptyset \vdash t : \forall \bar{X}.T} \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

By the induction hypothesis, there exists M' such that:

$$M', \emptyset \vdash t' : T \quad \text{and} \quad \vdash \mu' : M' \quad \text{and} \quad M \subseteq M'$$

Here, *we are stuck*. We would like to build a new instance of Gen, but we are missing $\bar{X} \mathrel{\#} M'$.

Mysterious slogan #2: *one must not generalize a type variable that might, after evaluation of the term, enter the store typing.* Aha!

This is what happens in the counter-example:

$$\text{let } x = \text{ref } (\lambda z.z : X \longrightarrow X) \text{ in } x := (\lambda z.z + 1); !x \text{ true}$$

The type variable X is generalized by GenLet. Yet, when ref $(\lambda z.z)$ reduces, $X \longrightarrow X$ becomes the type of the newly allocated cell, so it appears in the new store typing.

This is all well and good, but *how* do we enforce slogan #2? Should we somehow restrict $\bar{X}$ so as to ensure $\bar{X} \# M'$?

A number of rather complex historic approaches have been followed: see Leroy [1992] for a survey.

Then came Wright [1995], who suggested an amazingly simple solution, known as the *value restriction:* only values can be polymorphic.

$$\text{Gen}\quad \frac{M, \Gamma \vdash v : T \qquad \bar{X} \mathrel{\#} M, \Gamma}{M, \Gamma \vdash v : \forall \bar{X}.T}$$

The problematic proof case *vanishes:* we now never reduce under Gen. Subject Reduction holds again.

The problematic program is now ill-typed:

$$\text{let } x = \text{ref } (\lambda z.z) \text{ in } x := (\lambda z.z + 1); !x \text{ true}$$

Indeed, *ref (λz.z) is not a value*, so Gen is not applicable. The variable x must receive a monotype, but none is suitable.

With the value restriction, some pure programs become ill-typed, even though they were well-typed in the absence of references. This style of introducing references in ML is *not a conservative extension.*

This definition cannot receive a polymorphic type scheme:

$$\text{let } f = map\ id \qquad \text{list } T \rightarrow \text{list } T, \text{ for any type } T$$

A common work-around is to perform a manual η-expansion:

$$\text{let } f\ xs = map\ id\ xs \qquad \forall X.\text{list } X \rightarrow \text{list } X$$

In general, η-expansion is not semantics-preserving, so this must not be done blindly.

Experience has shown that *the value restriction is tolerable.* Even though it is not conservative, the search for better solutions has been pretty much abandoned.

In practice, the restriction is relaxed by delimiting a syntactic category of so-called *non-expansive terms* — terms whose evaluation definitely will not allocate new reference cells. Non-expansive terms form a strict superset of values.

Furthermore, Garrigue [2004] has relaxed the value restriction in a more subtle way.

Objective Caml implements both refinements.

(Most titles are clickable links to online versions.)

Aydemir, B., Charguéraud, A., Pierce, B., Pollack, R., and Weirich, S. 2008.
Engineering formal metatheory.
In *ACM Symposium on Principles of Programming Languages (POPL)*.

To appear.

Chlipala, A. 2008.
Parametric higher-order abstract syntax for mechanized semantics.

In *ACM International Conference on Functional Programming (ICFP)*. 143–156.

Bibliography]Bibliography

Crary, K., Weirich, S., and Morrisett, G. 2002.
Intensional polymorphism in type erasure semantics.
Journal of Functional Programming 12, 6 (Nov.), 567–600.

Gabbay, M. J. and Pitts, A. M. 2002.
A new approach to abstract syntax with variable binding.
Formal Aspects of Computing 13, 3–5 (July), 341–363.

Garrigue, J. 2004.
Relaxing the value restriction.
In *Functional and Logic Programming*. Lecture Notes in Computer Science, vol. 2998. Springer Verlag, 196–213.

[][

Harper, R. and Licata, D. R. 2007.
Mechanizing metatheory in a logical framework.
Journal of Functional Programming 17, 4–5, 613–673.

Hudak, P., Hughes, J., Peyton Jones, S., and Wadler, P. 2007.
A history of Haskell: being lazy with class.
In *ACM SIGPLAN Conference on History of Programming Languages*.

Leroy, X. 1992.
Typage polymorphe d'un langage algorithmique.
Ph.D. thesis, Université Paris 7.

Milner, R. 1978.
A theory of type polymorphism in programming.
Journal of Computer and System Sciences 17, 3 (Dec.), 348–375.

[][

Minamide, Y., Morrisett, G., and Harper, R. 1996.
Typed closure conversion.
In *ACM Symposium on Principles of Programming Languages (POPL)*. 271–283.

Mitchell, J. C. 1988.
Polymorphic type inference and containment.
Information and Computation 76, 2–3, 211–249.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999.
From system F to typed assembly language.
ACM Transactions on Programming Languages and Systems 21, 3 (May), 528–569.

[][

Pientka, B. 2007.
Proof pearl: The power of higher-order encodings in the logical framework LF.
In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Lecture Notes in Computer Science, vol. 4732. Springer Verlag, 246–261.

Pierce, B. C. 2002.
Types and Programming Languages.
MIT Press.

Pitts, A. M. 2000.
Parametric polymorphism and operational equivalence.
Mathematical Structures in Computer Science 10, 321–359.

[][

Pitts, A. M. 2006.
Alpha-structural recursion and induction.
Journal of the ACM 53, 459–506.

Pottier, F. and Rémy, D. 2005.
The essence of ML type inference.
In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, Chapter 10, 389–489.

Reynolds, J. C. 1974.
Towards a theory of type structure.
In *Colloque sur la Programmation*. Lecture Notes in Computer Science, vol. 19. Springer Verlag, 408–425.

[][

Reynolds, J. C. 1983.
Types, abstraction and parametric polymorphism.
In *Information Processing 83*. Elsevier Science, 513–523.

Strachey, C. 2000.
Fundamental concepts in programming languages.
Higher-Order and Symbolic Computation 13, 1–2 (Apr.), 11–49.

Tiuryn, J. and Urzyczyn, P. 2002.
The subtyping problem for second-order types is undecidable.
Information and Computation 179, 1, 1–18.

[][

Urban, C. and Tasson, C. 2005.
Nominal techniques in Isabelle/HOL.
In *International Conference on Automated Deduction (CADE)*. Lecture Notes in Computer Science, vol. 3632. Springer Verlag, 38–53.

Wadler, P. 1989.
Theorems for free!
In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*. 347–359.

Wadler, P. 2007.
The Girard-Reynolds isomorphism (second edition).
Theoretical Computer Science *375*, 1–3 (May), 201–226.

[][

Wells, J. B. 1995.
The undecidability of Mitchell's subtyping relation.
Technical Report 95-019, Computer Science Department, Boston University. Dec.

Wells, J. B. 1999.
Typability and type checking in system F are equivalent and undecidable.
Annals of Pure and Applied Logic 98, 1–3, 111–156.

[][

Wells, J. B. 2002.
The essence of principal typings.
In *International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 2380. Springer Verlag, 913–925.

Wright, A. K. 1995.
Simple imperative polymorphism.
Lisp and Symbolic Computation 8, 4 (Dec.), 343–356.

Wright, A. K. and Felleisen, M. 1994.
A syntactic approach to type soundness.
Information and Computation 115, 1 (Nov.), 38–94.