MPRI, Typage

Didier Rémy
(With much course meterial from François Pottier)

September 21, 2010

Plan of the course

Introduction

Simply-typed λ-calculus

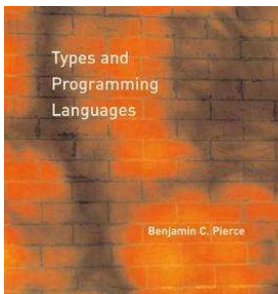
Polymorphism and System F

Polymorphism and System F

Reminder:

Course notes online at
http://gallium.inria.fr/~remy/mpri/
Also accessible from the MPRI url
http://mpri.master.univ-paris7.fr/C-2-4-2.html

See also Pierce [2002]:



This is the best choice for the two first lessons of the course.

Contents

- Why polymorphism?
- Polymorphic λ-calculus
- Type soundness
- Type erasing semantics
- Polymorphism and references
- Damas and Milner's type system

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:

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

which *instantiates* to the monomorphic types:

$$\begin{array}{c} (\text{int} \to \text{int} \to \text{bool}) \to \text{list int} \to \text{list int} \\ (\text{bool} \to \text{bool} \to \text{bool}) \to \text{list bool} \to \text{list bool} \\ \ldots \end{array}$$

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:

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

(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 seems almost free

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

$$(X_1 \to X_2) \to X_1 \to X_1 \to X_2 \times X_2$$

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

By saying that this term admits the polymorphic type:

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

we make polymorphism *internal* to the type system.

Towards type abstraction

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*.

Parametricity

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

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

reveals a *"free theorem"* about its behavior!

Basically, sorting commutes with (map f), provided f is order-preserving.

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 us 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.

There are 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:

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

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

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

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* (e.g. $2 + 2$), or, even worse, *upon its type derivation* (e.g. $\lambda x.\mathit{show}\,(\mathit{read}\,x)$).

Ad hoc versus parametric polymorphism: 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:

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

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

End of digression

Contents

- Why polymorphism?
- Polymorphic λ-calculus
- Type soundness
- Type erasing semantics
- Polymorphism and references
- Damas and Milner's type system

System F

The System F, (also known as: the polymorphic λ-calculus, the second-order λ-calculus; F_2) 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 the syntax and semantics of terms extended? There are several variants, depending on whether one adopts an *implicitly-typed* or *explicitly-typed* presentation of terms and a *type-passing* or a *type-erasing* semantics.

Explicitly-typed System F

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

$$\text{Tabs}\quad \frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T} \qquad \text{Tapp}\quad \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t\, T' : [X \mapsto T']T}$$

Terms are extended accordingly (we write F for the set of explicitly-typed terms)

$$t ::= \ldots \mid \Lambda X.t \mid t\, T$$

Type variables are explicitly bound and appear in type environments.

$$\Gamma ::= \ldots \mid \Gamma, X$$

Well-formedness of environment

Mandatory: We extend our previous convention to form environments, where Γ, X requires $X \# \Gamma$, *i.e.* X is neither in the domain nor in the image of Γ.

Optional: We may also require that environments be closed with respect to type variables, that is, $\mathrm{ftv}(T) \subseteq \mathrm{dom}(\Gamma)$ to form $\Gamma, x : T$.

We choose the stricter presentation where environment are closed.

The two definitions differ but in insignificant ways:

- The stricter definition allows fewer judgments, since judgments with open contexts are not allowed.
- However, they can always be closed by adding a prefix composed of a sequence of its free type variables.

Typing derivations of the loose presentation can always be closed to be strict.

Well-formedness of environments and types

Well-formedness of environments and types may also be defined (recursively) by inference rules:

WfEnvEmpty
$$\vdash \emptyset$$

WfEnvVar
$$\frac{\vdash \Gamma \qquad x \notin dom(\Gamma) \qquad \Gamma \vdash T}{\vdash \Gamma, x : T}$$

WfEnvTvar
$$\frac{\vdash \Gamma \qquad X \notin dom(\Gamma)}{\vdash \Gamma, X}$$

WfTypeVar
$$\frac{\vdash \Gamma \qquad X \in \Gamma}{\Gamma \vdash X}$$

WfTypeArrow
$$\frac{\Gamma \vdash T_1 \qquad \Gamma \vdash T_2}{\Gamma \vdash T_1 \to T}$$

WfTypeForall
$$\frac{\Gamma, X \vdash T}{\Gamma \vdash \forall X.T}$$

Well-formedness of environments and types

There is a choice whether well-formedness of environments should be made explicit or left implicit in typing rules.

Explicit well-formedness amounts to adding well-formedness premises to every rule where the environment or some type that appear in the conclusion did not appear in any premise.

Namely:

$$\text{Var}\ \frac{x : T \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : T} \qquad \text{Tapp}\ \frac{\Gamma \vdash t : \forall X.T \qquad \Gamma \vdash T'}{\Gamma \vdash t\ T' : [X \mapsto T']T}$$

Explicit well-formedness is more precise and better suited for mechanized proofs or for (more) complicated systems. However, it is a bit verbose and distracting. The two styles are equivalent.

We choose to leave it implicit. For documentation, we still indicate the well-formedness premises in the definition of typing rules.

Type-passing semantics

We need the following reduction for type-level expressions:

$$(\Lambda X.t)\ T \longrightarrow [X \mapsto T]t \tag{1}$$

Then, there is a choice.

The most common presentation of System F is that type abstraction stops the evaluation. It is described by:

$$\begin{aligned} v &::= \ldots \mid \Lambda X.t \\ E &::= \ldots \mid [\,]\ T \end{aligned}$$

However, this defines a type-passing semantics.

Indeed, $\Lambda X.((\lambda y : X.\, y)\ v)$ is a value while its type erasure is $(\lambda y.\, y)\ \lceil v \rceil$ is not—and can be further reduced.

Type-erasing semantics

We recover a type-erasing semantics if we choose to allow evaluation under type abstraction:

$$
\begin{aligned}
v &::= \ldots \mid \Lambda X.v \\
E &::= \ldots \mid [\,]\ T \mid \Lambda X.[\,]
\end{aligned}
$$

Then, we only need a weaker version of ι-reduction:

$$(\Lambda X.v)\ T \longrightarrow [X \mapsto T]v \qquad (\iota)$$

We now have:

$$\Lambda X.(\lambda y : X.\, y)\ v \longrightarrow \Lambda X.v$$

We will show below that this defines a type-erasing semantics.

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.

Type-passing versus type-erasing: *pros* and cons

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].

The *type-erasing* semantics, does not alter the semantics of untyped terms.

It also coincides with the semantics of ML—and, more generally, to the semantics of most programming languages. It also exhibits difficulties when adding side effects while the type-passing semantics does not.

Notice, that as a consequence of choosing a type-erasing semantics, we allow evaluation under a type abstraction.

In the following, we choose the type-passing semantics.

Reconciling type-passing and type-erasing views

If we restrict type abstraction to value-forms (which include values and variables), that is, we only allow $\Lambda X.t$ when t is a value-form, then the type-passing and type-erasing semantics coincide.

Indeed, under this restriction, *closed* type abstractions will always be type abstraction of values, and evaluation under type abstraction will never be used.

This restriction will be used when adding side-effects as a way to preserve type-soundness.

Explicitly-typed System F

We study the *explicitly-typed* presentation of System F first, because it is simpler to study.

Once, we have verified that the semantics is indeed type-preserving, many properties can be pull back to the implicitly-typed version, and in particular to its interesting ML subset.

Encoding data-structures

System F is quite expressive: it enables the encoding of data structures.

For instance, the church encoding of pairs is well-typed:

$$pair \triangleq \Lambda X_1.\Lambda X_2.\lambda x_1:X_1.\lambda x_2:X_2.\Lambda Y.\lambda y:X_1 \to X_2 \to Y.\ y\ x_1\ x_2$$

$$fst \triangleq \Lambda X_1.\Lambda X_2.\lambda y:\forall Y.(X_1 \to X_2 \to Y) \to Y.\ y\ X_1\ (\lambda x_1:X_1.\lambda x_2:X_2.x_1)$$

$$snd \triangleq \Lambda X_1.\Lambda X_2.\lambda y:\forall Y.(X_1 \to X_2 \to Y) \to Y.\ y\ X_2\ (\lambda x_1:X_1.\lambda x_2:X_2.x_2)$$

$$\lceil pair \rceil \triangleq \lambda x_1.\lambda x_2.\lambda y.\ y\ x_1\ x_2$$

$$\lceil fst \rceil \triangleq \lambda y.\ y\ (\lambda x_1.\lambda x_2.x_1)$$

$$\lceil snd \rceil \triangleq \lambda y.\ y\ (\lambda x_1.\lambda x_2.x_2)$$

Natural numbers, List, *etc.* can also be encoded.

Constructors and destructors

Unit, Pairs, Sums, *etc.* can also be added to System F as primitives.

We can then proceed as for simply-typed λ-calculus.

However, we may take advantage of the expressive type system of System F to deal with such extensions is a more elegant way: thanks to polymorphism, we need not add new typing rules for each extension.

We may instead add one typing rule for constants that is parametrized by an initial typing environment.

The allows sharing the meta-theoretical developments between the different extensions.

Let us first illustrate it with pairs.

Constructors and destructors — Pairs

Types are extended with a type constructor $\times$ of arity 2:

$$T ::= \ldots \mid T \times T$$

Expressions are extended with a constructor $(\cdot,\cdot)$ and two destructors proj_1 and proj_2 with the respective signatures:

$$\begin{aligned} Pair: &\quad \forall X. \forall Y. X \to Y \to X \times Y \\ proj_1: &\quad \forall X. \forall Y. X \times Y \to X \\ proj_2: &\quad \forall X. \forall Y. X \times Y \to Y \end{aligned}$$

which represent an initial environment Δ. We need not add any new typing rule, but instead type programs in the initial environment Δ.

This allows for the formation of partial applications of constructors and destructors. Hence, values are extended as follows:

$$\begin{aligned} v \quad ::= \quad \ldots \quad & \mid Pair \mid Pair\ T \mid Pair\ T\ T \mid Pair\ T\ T\ v \mid Pair\ T\ T\ v\ v \\ & \mid proj_i \mid proj_i\ T \mid proj_i\ T\ T \end{aligned}$$

Constructors and destructors

Pairs

We add the two following reduction rules:

$$proj_i\ T_1\ T_2\ (pair\ T_1'\ T_2'\ v_1\ v_2) \longrightarrow v_i \qquad (\delta_{pair})$$

Comments?

Constructors and destructors — Pairs

We add the two following reduction rules:

$$proj_i\ T_1\ T_2\ (pair\ T_1'\ T_2'\ v_1\ v_2) \longrightarrow v_i \qquad (\delta_{pair})$$

Comments?

- For well-typed programs, T_i and T_i' will always be equal, but the reduction will not check this at runtime.

  Instead, one could have defined the rule:

  $$proj_i\ T_1\ T_2\ (pair\ T_1\ T_2\ v_1\ v_2) \longrightarrow v_i \qquad (\delta'_{pair})$$

  The two semantics are equivalent on well-typed terms, but differ on ill-typed terms where δ'_{pair} may block when rule δ_{pair} would progress, ignoring type errors.
  With δ'_{pair}, the proof obligation in subject reduction will be simplified but replaced by a more stronger proof obligation in progress.

Constructors and destructors — Pairs

We add the two following reduction rules:

$$proj_i\ T_1\ T_2\ (pair\ T_1'\ T_2'\ v_1\ v_2) \longrightarrow v_i \qquad (\delta_{pair})$$

Comments?

- For well-typed programs, T_i and T_i' will always be equal, but the reduction will not check this at runtime.

  Instead, one could have defined the rule:

  $$proj_i\ T_1\ T_2\ (pair\ T_1\ T_2\ v_1\ v_2) \longrightarrow v_i \qquad (\delta'_{pair})$$

  The two semantics are equivalent on well-typed terms, but differ on ill-typed terms where δ'_{pair} may block when rule δ_{pair} would progress, ignoring type errors.
  With δ'_{pair}, the proof obligation in subject reduction will be simplified but replaced by a more stronger proof obligation in progress.

Constructors and destructors — Pairs

We add the two following reduction rules:

$$proj_i\ T_1\ T_2\ (pair\ T_1'\ T_2'\ v_1\ v_2) \longrightarrow v_i \qquad (\delta_{pair})$$

Comments?

- This presentation forces the programmer to specify the types of the components of the pair.
  However, since this is an explicitly type presentation, the types of the component is already known from the arguments of the pair (when present)
  This is should not be considered as a problem: explicitly-typed presentations are always verbose. Removing redundant type annotations is the task of type reconstruction.

Constructors and destructors — General case

Assume given a collection of type constructors F, with their arity ϱF. We assume that types respect the arities of type constructors.

A type $\mathsf{F}\,(\vec{T})$ is called and F-type. A *data type* is an F-type for some type constructor F.

Let Δ be an initial environment binding constants c of arity n to signatures of the form $\forall X._1 \ldots \forall X._k T_1 \rightarrow \ldots T_n$.

Constants are split into (value) constructors C or destructors d. We require that T_n be a data type whenever the constant is a constructor.

Constructors and destructors — General case

Expressions are extended with constants: Constants are typed as variables, but they types are looked up in Δ:

$$t ::= \ldots \mid c$$

$$\frac{c : T \in \Delta}{\Gamma \vdash c : T} \ \text{(Cst)}$$

Values are extended with partial or full applications of constructors and partial applications of destructors:

$$\begin{array}{rcll} v & ::= & \ldots & \\ & \mid & C\, T_1 \ldots T_p\, v_1 \ldots v_q & q \le \varrho C \\ & \mid & d\, T_1 \ldots T_p\, v_1 \ldots v_q & q < \varrho d \end{array}$$

For each destructor d of arity n, we assume given a set of δ-rules of the form

$$d\, T_1 \ldots T_k\, v_1 \ldots v_n \longrightarrow t \qquad (\delta_d)$$

Constructors and destructors — General case

Of course, we need assumptions to relate typing and reduction of constants:

Subject-reduction for constants: δ-rules preserve typings for well-typed terms: If $\vec{X} \vdash t_1 : T$ and $t_1 \longrightarrow_\delta t_2$ then $\vec{X} \vdash t_2 : T$.

Progress for constants: If $\vec{X} \vdash t_1 : T$ and t_1 is of the form $d\ T_1\ \ldots\ T_k\ v_1\ \ldots\ v_n$ where $n = \varrho d$, then there exists t_2 such that $t_1 \longrightarrow t_2$.

Intuitively, progress means that the domain of destructors is at least as large as specified by their type in Δ.

Example — Unit, Pairs

Adding units:

- Introduce a type constant unit
- Introduce a constructor () of arity 0 of type unit.
- No primitive and no reduction rule is added.

The assumptions obviously hold in the absence of destructors.

The previous example of pairs fits exactly in this framework.

Example

Fixpoint

We introduce a destructor

$$\text{fix} : \forall X.\, \forall Y.\, ((X \to Y) \to X \to Y) \to X \to Y \qquad \in \Delta$$

of arity 2, together with the δ-rule

$$\text{fix}\; T_1\; T_2\; v_1\; v_2 \longrightarrow v_1\; (\text{fix}\; T_1\; T_2\; v_1)\; v_2 \qquad (\delta_{\text{fix}})$$

It is straightforward to check the assumptions:

- Progress is by construction, since fix does not destruct values.
- Subject reduction is straight-forward. Assume that $\Gamma \vdash \text{fix}\; T_1\; T_2\; v_1\; v_2 : T$. By inversion of typing rules, T must be equal to T_2, v_1 and v_2 must be of types $(T_1 \to T_2) \to T_1 \to T_2$ and T_1 in the typing context Γ. We may then easily build a derivation of the judgment $\Gamma \vdash v_1\; (\text{fix}\; T_1\; T_2\; v_1)\; v_2 : T$

Contents

- Why polymorphism?
- Polymorphic λ-calculus
- Type soundness
- Type erasing semantics
- Polymorphism and references
- Damas and Milner's type system

Type soundness

The structure of the proof is similar to the case of simply-typed λ-calculus and follows from subject reduction and progress.

Subject reduction uses the following lemmas:

- permutation and weakening
- expression substitution
- type substitution
- compositionality

Type soundness — Weakening

Lemma (Weakening)

Assume $\Gamma \vdash t : T$.
1) If $x \# \Gamma$, then $\Gamma, x : T' \vdash t : T$.
2) If $Y \# \Gamma$, then $\Gamma, Y \vdash t : T$.

Case 1) is as for simply-typed λ-calculus. Case 2) is new for System F, since now environments also introduce type variables. The proof schema is similar to the case of simply-typed λ-calculus. We just have more cases. We still reason by induction on t, then by cases on t applying the inversion lemma (for System F).

Cases for value and type abstraction appeal to the permutation lemma, which must also be extended.

Lemma (Permutation)

If $\Gamma, \Delta, \Delta', \Gamma' \vdash t : T$ and $\Delta \# \Delta'$, then $\Gamma, \Delta', \Delta, \Gamma' \vdash t : T$.

Type Soundness — Type substitution

Lemma (Expression substitution, strengthened)

If $\Gamma, x : T, \Gamma' \vdash t' : T'$ *and* $\Gamma \vdash t : T$ *then* $\Gamma \vdash [x \mapsto t]t' : T'$.

The proof is similar to that for the simply-typed λ-calculus, with just a few more cases.

We have strengthened the lemma with an arbitrary context Γ' as for the simply-typed λ-calculus

We have also generalized the lemma with an arbitrary context Γ on the left and an arbitrary expression t, as this does not complicate the proof.

Type Soundness — Type substitution

Lemma (Type substititition , strengthened)

If $\Gamma, X, \Gamma' \vdash t : T'$ *then* $\Gamma, [X \mapsto T]\Gamma' \vdash [X \mapsto T]t : [X \mapsto T]T'$.

The proof is by induction on t.

The interesting cases are for type and value abstraction, which required the strengthened version with an arbitrary typing context Γ' on the right. Then, the proof is straightforward.

We also generalized the lemma using an arbitrary environment instead of the empty environment, as it does not change the proof.

Type Soundness — Subject reduction

Proof of subject reduction.

The proof is by induction on t.

Using the previous lemmas it is straightforward.

Interestingly, the case for δ-rules follows from the subject-reduction assumptions for constants. □

Type soundness — Progress

Progress is restated as follows:

Theorem (Progress, strengthened)

A well-typed, irreducible closed term is a value:
if $\vec{X} \vdash t : T$ and $t \not\longrightarrow$, then t is some value v.

The theorem has been strengthened, using a sequence of type variables $\vec{X}$ for the typing context instead of the empty environment.

It is then proved by induction and case analysis on t.

It relies mainly on the classification lemma (reminded below) and the progress assumption for destructors.

Type soundness — Classification

The classification lemma is slightly modified to account for polymorphic types and constructed types.

Lemma (Classification)

Assume $\bar{X} \vdash v : T$

- *If T is an arrow type, then v is either a function or a partial application of a constant.*
- *If T is a polymorphic type, then v is either a type abstraction of a value or a partial application of a constant to types.*
- *If T is a constructed type, then v is constructed value.*

The last case can be refined by partitioning constructors into their associated type-constructor: If T is a F-constructed type, then v is a value constructed with a F-constructor.

Normalization

Theorem

Reduction terminates in pure System F.

This is also true for arbitrary reductions and not just for call-by-value reduction.

This is a difficult proof, which generalizes the proof method for the simply-typed λ-calculus. It is due to Girard [1972] (See also Girard et al. [1990]).

Contents

- Why polymorphism?
- Polymorphic λ-calculus
- Type soundness
- Type erasing semantics
- Polymorphism and references
- Damas and Milner's type system

Implicitly-typed System F

The syntax and dynamic semantics of terms are that of the untyped λ-calculus. However, we only accept terms that are the type erasure of a term in F.

We write $\lceil F \rceil$ for the set of implicitly-typed terms, and use letters a, w, and D to range over implicitly-typed terms, values and evaluation contexts.

We may equivalently rewrite the typing rules to operate directly on unannotated terms by dropping all type information in terms, as we did for the simply-typed λ-calculus. The two new rules are:

Tabs
$$\frac{\Gamma, X \vdash a : T}{\Gamma \vdash a : \forall X.T}$$

Tapp
$$\frac{\Gamma \vdash a : \forall X.T'}{\Gamma \vdash a : [X \mapsto T]T'}$$

As a consequence, the rules that introduce and eliminate the universal quantifier are non-syntax-directed.

Implicitly-typed System F On the side condition $X \# \Gamma$

Notice that the explicit introduction of variable X in the premise of Rule Tabs contains an implicit side condition $X \# \Gamma$ due to the assumption on the formation of contexts.

In implicitly-typed System F, we could also erase type declarations in typing context. (Although, in some extensions of System F, type variable may carry a kind or a bound must be explicitly introduced.)

Then, we would need an explicit side-condition on Rule Tabs:

Tabs-Bis

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

Why is the side condition important?...

Implicitly-typed System F — On the side condition $X \# \Gamma$

Omitting the side condition leads to *unsoundness:*

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : X_1 \vdash x : X_1}\textsc{Var}}{x : X_1 \vdash x : \forall X_1.X_1}\textsc{Broken Tabs}}{x : X_1 \vdash x : X_2}\textsc{Tapp}}{\varnothing \vdash \lambda x.\, x : X_1 \rightarrow X_2}\textsc{Abs}}{\varnothing \vdash \lambda x.\, x : \forall X_1.\forall X_2.X_1 \rightarrow X_2}\textsc{Tabs-Bis}
$$

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

Implicitly-typed System F — On the side condition $X \# \Gamma$

The is equivalent to using ill-formed context:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : X_1, X_1 \vdash x : X_1}\;\text{Broken Var}}{x : X_1 \vdash x : \forall X_1.X_1}\;\text{Broken Tabs}}{x : X_1 \vdash x : X_2}\;\text{Tapp}}{\varnothing \vdash \lambda x{:}X_1.\, x : X_1 \to X_2}\;\text{Abs}}{\varnothing \vdash \Lambda X_1.\Lambda X_2.\lambda X_1{:}x.\, x : \forall X_1.\forall X_2.X_1 \to X_2}\;\text{Tabs}
$$

Implicitly-typed System F — On the side condition $X \# \Gamma$

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

In that view, Tabs-Bis corresponds to the axiom:

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

Type-erasing typechecking

Type systems for implicitly-typed System F and explicitly type System F coincide.

Lemma

$\Gamma \vdash a : T$ in implicitly-typed System F if and only if there exists an explicitly-typed expression t whose erasure is a such that $\Gamma \vdash t : T$.

Trivial.

We write F and $\lceil F \rceil$ for the explicitly-typed and implicit-typed versions of System F.

An example $\lambda fxy.(f\ x, f\ y)$

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?

An example $\lambda fxy.(f\,x, f\,y)$

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: ...

Incomparable types in System F

$\lambda fxy.\,(f\ x, f\ y)$

Which of the two is more general?

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

Incomparable types in System F $\lambda fxy.(f\ x, f\ y)$

Which of the two is more general?

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

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!)

Notions of instance in $[F]$

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 must 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...

Syntactic notions of instance in $[F]$

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

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

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 *derivable* in the implicitly-typed version:

Sub
$$\frac{\Gamma \vdash a : T_1 \qquad T_1 \leq T_2}{\Gamma \vdash a : T_2}$$

(A direct proof may not be so easy exercise: prove it!)

Syntactic notions of instance in F

What is the counter-part of instance in explicitly-typed System F?

Assume $\Gamma \vdash t : T_1$ and $T_1 \leq T_2$. How can we see t with type T_2?

Well, t_1 and t_2 must be of the form $\forall \bar{X}.T$ and $\forall \bar{Y}.[\vec{X} \mapsto \vec{T}]T$ where $\bar{Y} \# \forall \bar{X}.T$. *W.l.o.g*, we may assume that $\bar{Y} \# \Gamma$.

We can wrap t with a *retyping context*, as follows.

$$\left.\text{Tabs}^*\dfrac{\text{Tapp}^*\dfrac{\text{Weak.}\dfrac{\Gamma \vdash t : \forall \vec{X}.T \qquad \bar{Y} \# \Gamma \;\; (1)}{\Gamma, \vec{Y} \vdash t : \forall \vec{X}.T}}{\Gamma, \vec{Y} \vdash t\;\vec{T} : [\vec{X} \mapsto \vec{T}]T}}{\Gamma \vdash \Lambda \vec{Y}.t\;\vec{T} : \forall \vec{Y}.[\vec{X} \mapsto \vec{T}]T}\right\}$$

Admissible rule:

$$\text{Sub}\;\dfrac{\Gamma \vdash t : \forall \vec{X}.T \qquad \bar{Y} \# \forall \vec{X}.T \;\; (2)}{\Gamma \vdash \Lambda \vec{Y}.t\;\vec{T} : \forall \vec{Y}.[\vec{X} \mapsto \vec{T}]T}$$

If condition (2) holds, condition (1) may always be satisfied up to a renaming of $\bar{Y}$.

Retyping contexts in F

In F, subtyping is a judgment $\Gamma \vdash T_1 \leq T_2$ to track well-formedness of types. Subtyping relations can be witnessed by retyping contexts.

Retyping contexts are just wrapping type abstractions and type applications around expressions, without changing their type erasure.

$$C ::= [] \mid \Lambda X.C \mid C\ T$$

(Notice that C are arbitrarily *deep*, as opposed to evaluation contexts)

Let us write $\Gamma \vdash C[T_1] : T_2$ iff $\Gamma, x : T_1 \vdash C[x] : T_2$.

If $\Gamma \vdash t : T_1$ and $\Gamma \vdash C[T_1] : T_2$, then $\Gamma \vdash C[t] : T_2$,

Then $\Gamma \vdash T_1 \leq T_2$ iff $\Gamma \vdash C[T_1] : T_2$. for some retyping context C.

In System F, retyping contexts can only change "toplevel" polymorphism. In particular, they cannot weaken the result types of functions or strengthen their arguments.

Another syntactic notion of instance: System F_η

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

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

Distributivity
$$\forall X.\,(T_1 \to T_2) \leq (\forall X.\,T_1) \to (\forall X.\,T_2)$$

Congruence-→
$$\frac{T_2 \leq T_1 \qquad T_1' \leq T_2'}{T_1 \to T_1' \leq T_2 \to T_2'}$$

Congruence-∀
$$\frac{T_1 \leq T_2}{\forall X.T_1 \leq \forall X.T_2}$$

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

A typing of an expression t is a pair Γ, T such that $\Gamma \vdash t : T$.

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

> *Every well-typed term t admits a principal typing — one whose instances are exactly the typings 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?

The *simply-typed λ-calculus has principal typings,* with respect to a substitution-based notion of instance (See lesson on type inference).

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 and have yet to see a practical application.

A weaker property is to have *principal types*. Given an environment Γ and an expression t is there a type T for t in Γ such that all other types of t in Γ are instances of T.

Damas and Milner's type system (coming up next) does not have *principal typings* but it has *principal types* and *decidable type inference*.

Type soundness for $\lceil F \rceil$

Subject reduction and progress imply the soundness of the *explicitly*-typed System F. What about the *implicitly*-typed version?

Can we reuse the soundness proof for the explicitly-typed version? Can we pullback subject reduction and progress from F to $\lceil F \rceil$?

Progress? Given a well-typed term $a \in \lceil F \rceil$, can we find a term $t \in F$ whose erasure is a and since t is a value or reduces, conclude that a is a value or reduces?

Subject reduction? Given a well-typed term $a_1 \in \lceil F \rceil$ of type T that reduces to a_2, can we find a term $t_1 \in F$ whose erasure is a_1 and show that t_1 reduces to a term t_2 whose erasure is a_2 to conclude that the type of a_2 is the type a_1?

In both cases, this reasoning requires a *type-erasing* semantics.

Type erasing semantics

We claimed earlier that the explicitly-typed System F has an erasing semantics. We now verify it.

There is a difference with the simply-typed λ-calculus because the reduction of type applications on explicitly-typed terms is dropped on implicitly-typed, hence the two reductions do not coincided *exactly*.

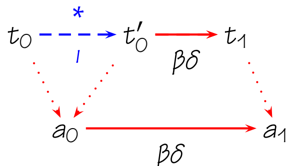
The way to formalize this is to split reduction steps into $\beta\delta$-steps which they originate from β or δ rules and are preserved by type-erasure, and ι-steps which originate from the reduction of type applications and disappear by type-erasure:

Type erasing semantics

We claimed earlier that the explicitly-typed System F has an erasing semantics. We now verify it.

There is a difference with the simply-typed λ-calculus because the reduction of type applications on explicitly-typed terms is dropped on implicitly-typed, hence the two reductions do not coincided *exactly*.

The way to formalize this is to split reduction steps into $\beta\delta$-steps which they originate from β or δ rules and are preserved by type-erasure, and ι-steps which originate from the reduction of type applications and disappear by type-erasure:

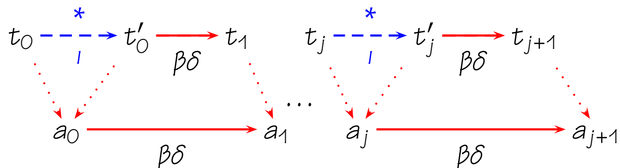


Type erasing semantics

We claimed earlier that the explicitly-typed System F has an erasing semantics. We now verify it.

There is a difference with the simply-typed λ-calculus because the reduction of type applications on explicitly-typed terms is dropped on implicitly-typed, hence the two reductions do not coincided *exactly*.

The way to formalize this is to split reduction steps into $\beta\delta$-steps which they originate from β or δ rules and are preserved by type-erasure, and ι-steps which originate from the reduction of type applications and disappear by type-erasure:

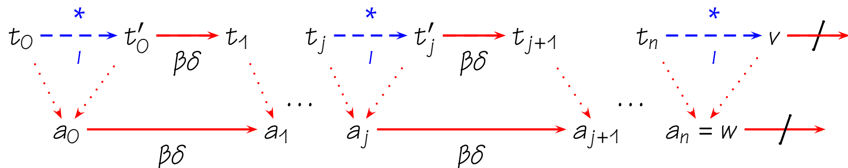


Type erasing semantics

We claimed earlier that the explicitly-typed System F has an erasing semantics. We now verify it.

There is a difference with the simply-typed λ-calculus because the reduction of type applications on explicitly-typed terms is dropped on implicitly-typed, hence the two reductions do not coincided *exactly*.

The way to formalize this is to split reduction steps into $\beta\delta$-steps which they originate from β or δ rules and are preserved by type-erasure, and ι-steps which originate from the reduction of type applications and disappear by type-erasure:



Type erasing semantics

Of course, the semantics can only be type-erasing if δ-rules do not themselves depend on type information.

We assume, that $t_1 \longrightarrow_\delta t_2$ and $\lceil t'_1 \rceil = \lceil t_1 \rceil$ implies that there exists t'_2 such that $t'_1 \longrightarrow_\delta t'_2$ and $\lceil t'_2 \rceil = \lceil t_2 \rceil$.

This implies that the type erasure of δ-reduction on explicitly-typed terms defines a deterministic relation on implicitly-typed terms.

Type erasing semantics — Forward simulation

Type erasure simulates in $\lceil F \rceil$ the reduction in F upto ι-steps:

Lemma (Forward simulation)

Assume $\Gamma \vdash t_1 : T$.

1) *If* $t_1 \longrightarrow_\iota t_2$, *then* $\lceil t_1 \rceil = \lceil t_2 \rceil$

2) *If* $t_1 \longrightarrow_{\beta\delta} t_2$, *then* $\lceil t_1 \rceil \longrightarrow_{\beta\delta} \lceil t_2 \rceil$

Both paths are easy by definition of type erasure.

Type erasing semantics Backward simulation

The backward direction is more delicate, since there are usually many expressions of F whose erasure is a given expression in $\lceil F \rceil$, as $\lceil \cdot \rceil$ is not injective.

Lemma (Backward simulation)

Assume $\Gamma \vdash t_1 : T$ and $\lceil t_1 \rceil \longrightarrow a$. Then, there exists a term t_2 such that $t_1 \longrightarrow_\iota^* \longrightarrow_{\beta\delta} t_2$ and $\lceil t_2 \rceil = a$.

The lemma can first be shown assuming that t_1 is ι-normal.

The general case follows, since then t_1 ι-reduces to a normal form t_1' which preserve typings, hence the lemma can be applied to t_1' instead of t_1.

Notice that we use the termination of ι-reduction. Indeed, if ι-reduction could diverge, the semantics would not be type-erasing.

Type erasing semantics — Backward simulation

We will use the following observations:

1) A term whose erasure is $D[a]$ is of the form $\vec{E}[t_1]$ where $\lceil \vec{E} \rceil$ and $\lceil t \rceil$ are D and a, and moreover, t does not start with a type abstraction nor a type application.
2) An evaluation context $\vec{E}$ whose erasure is the empty context is a retyping context C.
3) If $C[t]$ is in ι-normal form, then C is of the form $\Lambda \vec{X}.[]\ \vec{T}$.

Type erasing semantics — Backward simulation

Lemma (inversion of type erasure)

Assume $\lceil t \rceil = a$

- *If* a *is* x, *then* t *is* $C[x]$
- *If* a *is* c, *then* t *is* $C[c]$
- *If* a *is* $\lambda x{:}a_1.$, *then* t *is* $C[\lambda x{:}t_1.]$ *with* $\lceil t_1 \rceil = a_1$
- *If* a *is* $a_1\ a_2$, *then* t *is* $C[t_1\ t_2]$ *with* $\lceil t_i \rceil = a_i$

The proof is by induction on t.

Type erasing semantics — Backward simulation

Lemma (Inversion of type erasure for well-typed values)

Assume $\Gamma \vdash t : T$ and t is ι-normal. If $\lceil t \rceil$ is a value w, then t is a value v. Moreover,

- *If w is $\lambda x. a_1$, then v is $\Lambda \vec{X}.\lambda x{:}T.t_1$ with $\lceil t_1 \rceil = a_1$.*
- *If w is a partial application $c\, w_1 \ldots w_n$ then v is $C[c\, \vec{T}\, v_1 \ldots v_n]$ with $\lceil v_i \rceil = w_i$.*

The proof is by induction on t. It uses the inversion of type erasure, then analysis of the typing derivation to restrict the form of retyping contexts.

Type erasing semantics — Backward simulation

The proof of backward simulation in the case t is ι-normal is by induction on the reduction in $[F]$.

Case $\lceil t \rceil$ is $(\lambda x. a_1)\, w$:

Then, t is of the form $C_1[C_2[\lambda x{:}T_1.t_1]\, t_2]$ where $\lceil t_1 \rceil$ is a_1 and $\lceil t_2 \rceil$ is w. Since C_1 is an evaluation context, $C_2[\lambda x{:}T_1.t_1]$ is ι-normal. Hence C_2 is of the form $\Lambda \vec{X}.[]$. Since $C_2[\lambda x{:}T_1.t_1]$ is applied, it must have an arrow type, hence $\vec{X}$ must be empty. Hence, C_2 is empty and we are left with $C_1[(\lambda x{:}T_1.t_1)\, t_2]$.

Since t_2 is an evaluation position, it is in ι-normal form. Since its erasure is a value w, it should itself be a value v.

Therefore, $(\lambda x{:}T_1.t_1)\, v$ β-reduces to $[x \mapsto v]t_1$ and t reduces to $C_1[[x \mapsto v]t_1]$ whose erasure is $[x \mapsto v]a_1$, *i.e.* a.

Type erasing semantics — Backward simulation

Case $\lceil t \rceil$ is $D[a_1]$ and $a_1 \longrightarrow a_2$:

Then, t is of the form $\vec{E}[t_1]$ where $\lceil \vec{E} \rceil$ and $\lceil t_1 \rceil$ are D and a_1.

By compositionality, t_1 is well-typed.

By induction hypothesis, it $\beta\delta$-reduces to a term t_2 whose erasure is a_2.

Hence, t reduces to $\vec{E}[t_2]$ whose erasure is $D[a_2]$, *i.e.* a.

Type erasing semantics — Backward simulation

Case $\lceil t \rceil$ is a partial application $(d\, w_1 \ldots w_n)$ and reduces to a:

Then, t is of the form $C[d\, \vec{T}\, v_1\, \ldots v_n]$ where $\lceil v_i \rceil$ is w_i.

By the progress assumption on constants, the term $(d\, \vec{T}\, v_1 \ldots v_n)$ must δ-reduce to some term t'. Hence, t reduces to $C[t']$.

Since the erasure of $d\, \vec{T}\, v_1 \ldots v_n$ is $d\, w_1 \ldots w_n$, then by the assumption that constants have a type-erasing semantics. the erasure of t', which is also the erasure of $C[t']$, must be a.

Type erasing semantics — On bisimulations

Using bisimulations to show that compilation preserves the semantics given in small-step style is a classical technique.

For example, this technique is *heavily* used in the CompCert project to prove the correctness of a C-compiler to assembly code in Coq, using a dozen of successive intermediate languages.

Type soundness for implicitly-typed System F

We may now easily transpose subject reduction and progress from the implicitly-typed version to the implicitly-typed version of System F.

Progress Well-typed expressions in $\lceil F \rceil$ have an antecedent in ι-normal form F, which, by progress in F, either $\beta\delta$-reduces or is a value, hence whose type erasure reduces or is a value.

Subject reduction Assume that $\Gamma \vdash a_1 : T$ and $a_1 \longrightarrow a_2$. There exists a term t in ι-normal form such that $\Gamma \vdash t_1 : T$ and $t_1 \longrightarrow t_2$ and $\lceil t_2 \rceil$ is a_2. By subject reduction in F, $\Gamma \vdash t_2 : T$; by type erasure $\Gamma \vdash a_2 : T$.

Type erasing semantics

The design of advanced typed systems for programming languages is usually done in explicitly-typed version, but with a type-erasing semantics in mind, but this is not always checked in details.

While the forward simulation is usually straightforward, the backward simulation is often harder. As the type systems gets more complicated, reduction at the level of types also gets more complicated.

It is important and not always obvious that type reduction terminates and is rich enough to never block reductions that could occur in the type erasure.

Contents

- Why polymorphism?
- Polymorphic λ-calculus
- Type soundness
- Type erasing semantics
- Polymorphism and references
- Damas and Milner's type system

Combining extensions

We have shown how to extend simply-typed λ-calculus with

- polymorphism, and
- references, in a previous session.

Can we combine these two extensions?

Beware of polymorphic locations!

When adding references, we noted that type soundness relies on the fact that *every reference cell (or memory location) has a fixed type*.

Otherwise, if a location had two types $\mathsf{ref}\, T_1$ and $\mathsf{ref}\, T_2$, one could store a value of type T_1 and read back a value of type T_2.

Hence, it should also be unsound if a location could have type $\forall X. \mathsf{ref}\, T$ (where X appears in T) as it could then be specialized to both types $\mathsf{ref}\, [X \mapsto T_1]T$ and $\mathsf{ref}\, [X \mapsto T_2]T$.

By contrast, a location ℓ can have type $\mathsf{ref}\, (\forall X. T)$: this says that ℓ stores values of polymorphic type $\forall X. T$, but ℓ, as a value, is viewed with the monomorphic type $\mathsf{ref}\, (\forall X. T)$.

A counter example

Still, if naively extended with references, System F allows construction of polymorphic references, which breaks subject reduction:

A counter example

Still, if naively *extended* with references, System *F* allows construction of polymorphic references, which breaks subject reduction:

$$
\begin{array}{ll}
& \text{let } y : \forall X.\, \text{ref } (X \to X) = \Lambda x.\text{ref } (\lambda z{:}X.\, z) \text{ in} \\
& \quad (y \text{ bool}) := \mathit{not}; \\
& \quad !(y\ (\text{int}))\ 1 / \varnothing \\
\xrightarrow{*} & \mathit{not}\ 1 / \ell \mapsto \mathit{not}
\end{array}
$$

A counter example

Still, if *naively extended* with references, System *F* allows construction of polymorphic references, which breaks subject reduction:

$$
\begin{array}{ll}
 & \text{let } y : \forall X.\, \text{ref } (X \to X) = \Lambda x.\text{ref } (\lambda z{:}X.\, z) \text{ in} \\
 & \quad (y \text{ bool}) := \mathit{not}; \\
 & \quad !(y\ (\text{int}))\ 1/\varnothing \\
\xrightarrow{*} & \mathit{not}\ 1/\ell \mapsto \mathit{not}
\end{array}
$$

What happens is that the evaluation of the reference:

- creates and returns a location ℓ bound to the identity function $\lambda z{:}X.\,z$ of type $X \to X$,
- abstracts X in the result and binds it to y with the polymorphic type $\forall X.\, X \to X$;
- writes the location at type $\text{bool} \to \text{bool}$ and reads it back at type $\text{int} \to \text{int}$.

Nailing the bug

In the counter-example, the first reduction step uses the following rule (where v is $\lambda x{:}X.\,x$ and T is $X \to X$).

$$\text{Context}\ \frac{\text{ref}\ v/\varnothing \longrightarrow \ell/\ell \mapsto v}{\Lambda X.\text{ref}\ v/\varnothing \longrightarrow \Lambda X.\ell/\ell \mapsto v}$$

While we have

$$X \vdash \text{ref}\ v/\varnothing : \text{ref}\ T \qquad \text{and} \qquad X \vdash \ell/\ell \mapsto v : \text{ref}\ T$$

We have

$$\vdash \Lambda X.\text{ref}\ v/\varnothing : \forall X.\,\text{ref}\ T \qquad \text{but not} \qquad \vdash \Lambda X.\ell/\ell \mapsto v : \forall X.\,\text{ref}\ T$$

Hence, the context case of subject reduction breaks.

Nailing the bug

The typing derivation of $\Lambda X.\ell$ requires a store typing M of the form $\ell : T$ and a derivation of the form:

$$\text{Tabs}\ \frac{M, X \vdash \ell : \text{ref}\ T}{M \vdash \Lambda X.\ell : \forall X.\, \text{ref}\ T}$$

However, the typing context M, X is ill-formed as X appears free in M.

Instead, a well-formed premise should bind X earlier as in $X, M \vdash \ell : \text{ref}\ T$, but then, Rule Tabs cannot be applied.

By contrast, the expression $\text{ref}\ v$ is pure, so M may be empty:

$$\text{Tabs}\ \frac{X \vdash \text{ref}\ v : \text{ref}\ T}{\varnothing \vdash \text{ref}\ v : \forall X.\, \text{ref}\ T}$$

The expression is $\Lambda X.\ell$ is correctly rejected as ill-typed, so $\Lambda X.(\text{ref}\ v)$ should also be rejected.

Fixing the bug

Mysterious slogan:

> *One must not abstract over a type variable that might, after evaluation of the term, enter the store typing.*

Indeed, this is was happens in our example. The type variable X which appears in the type of v is abstracted in front of $\mathsf{ref}\, v$.

When $\mathsf{ref}\, v$ reduces, $X \rightarrow X$ becomes the type of the fresh location ℓ, which appears in the new store typing.

This is all well and good, but *how* do we enforce this slogan?

Fixing the bug

In the context of ML, 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 value-forms can be abstracted over.

TAbs

$$\frac{\Gamma, X \vdash u : T}{\Gamma \vdash \Lambda X.u : \forall X.T}$$

Value forms:

$$u ::= x \mid v \mid \Lambda T.u \mid u\ T$$

The problematic proof case *vanishes*, as we now never reduce under type abstraction. The form $\Lambda X.E$ of evaluation context becomes useless and can be removed.

Subject reduction holds again.

A good intuition: internalizing configurations

A configuration t/μ is an expression t in a memory μ. The memory can be viewed as a recursive extensible record.

The configuration t/μ may be viewed as the recursive definition (of values) $\text{let } rec\ m : M = \mu \text{ in } [\ell \mapsto m.\ell]t$ where M is a store typing for μ.

The store typing rules are coherent with this view.

Allocation of a reference is a reduction of the form

$$\begin{array}{lll} & \text{let } rec\ m : M & = \mu \quad \text{in } E[\text{ref } T\ v] \\ \longrightarrow & \text{let } rec\ m : M, \ell : T & = \mu, \ell \mapsto v \text{ in } E[m.\ell] \end{array}$$

For this transformation to preserve well-typedness, it is clear that the evaluation context must not bind any type variable appearing in T.

Otherwise, we are violating the scoping rules.

Clarifying the typing rules

Let us review the typing rules for configurations:

Config
$$\frac{\vec{X} \vdash t : T \qquad \vec{X} \vdash \mu : M}{\vec{X} \vdash t/\mu : T}$$

Store
$$\frac{\forall \ell \in \mathrm{dom}(\mu), \quad \vec{X}, M, \varnothing \vdash \mu(\ell) : M(\ell)}{\vec{X} \vdash \mu : M}$$

Because we explicitly introduce type variables in judgments, closed configurations must be typed in an environment composed of type variables.

Because we never reduce under type abstraction, these variables need not be changed during evaluation and can be placed in front of the store typing.

Clarifying the typing rules

Judgments are now of the form $\vec{X}, M, \Gamma \vdash t : T$ although we may see $\vec{X}, M, \Gamma$ as a whole typing context Γ'.

For locations, we need a new context formation rule:

WfEnvLoc

$$\frac{\vdash \Gamma \qquad \Gamma \vdash T \qquad \ell \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \ell : T}$$

This allows locations to appear anywhere. However, in a derivation of a closed term, the typing context will always be of the form $\vec{X}, M, \Gamma$ where:

- M only binds locations (to arbitrary types) and
- Γ does not bind locations.

Clarifying the typing rules

The typing rule for memory locations (where Γ is of the form $\vec{X}, M, \Gamma'$)

$$\frac{\text{Loc}}{\Gamma \vdash \ell : \mathsf{ref}\ \Gamma(\ell)}$$

In System F, typing rules for references need not be primitives.
We may instead treat them as constants of the following types:

$$\begin{aligned} \mathsf{ref} &: \quad \forall X. X \rightarrow \mathsf{ref}\ X \\ (!) &: \quad \forall X. \mathsf{ref}\ X \rightarrow X \\ (:=) &: \quad \forall X. \mathsf{ref}\ X \rightarrow X \rightarrow \mathsf{unit} \end{aligned}$$

There are all destructors (event ref) with the obvious arities.

The δ-rules are adapted to carry explicit type parameters:

$$\begin{aligned} \mathsf{ref}\ T\ v / \mu &\longrightarrow \ell / \mu[\ell \mapsto v] \qquad \text{if } \ell \notin dom(\mu) \\ (:=)\ T\ \ell\ v / \mu &\longrightarrow () / \mu[\ell \mapsto v] \\ (!)\ T \ell / \mu &\longrightarrow \mu(\ell) / \mu \end{aligned}$$

Stating type soundness

Lemma

δ-rules preserve well-typedness of closed configurations.

Theorem (Subject reduction)

Reduction of closed configurations preserves well-typedness.

Lemma

A well-typed closed configuration t/μ where t is a full application of constants ref, (!), and (:=) to types and values can always be reduced.

Theorem (Progress)

A well typed irreducible closed configuration t/μ is a value.

Consequences

The problematic program is now syntactically ill-formed:

$$\begin{array}{l} \text{let } y : \forall X.\, \text{ref}\,(X \to X) = \Lambda x.\text{ref}\,(\lambda z{:}X.\,z) \text{ in} \\ \quad (:=)\,(\text{bool} \to \text{bool})\,(y\ \text{bool})\ \mathit{not}; \\ \quad !\,(\text{int} \to \text{int})\,(y\,(\text{int}))\ 1 \end{array}$$

Indeed, $\text{ref}\,(\lambda z{:}X.\,z)$ is not a value, but the application of a unary destructor to a value.

Consequences

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 System F (or in ML) is *not a conservative extension.*

Assuming:

$$map : \forall X.\, \text{list}\ X \to \text{list}\ X \qquad\qquad id : \forall X.\, X \to X$$

This expression is ill-typed:

$$\Lambda X. map\ X\ (id\ X)$$

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

$$\Lambda X. \lambda y : \text{list}\ X.\, map\ X\ (id\ X)\ y$$

Of course, in the presence of side effects, η-expansion is *not* semantics-preserving, so this must not be done blindly.

In practice

The value restriction can be slightly relaxed by enlarging the class of value-forms to 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 value-forms.

Garrigue [2004] relaxes the value restriction in a more subtle way, which is justified by a subtyping argument.

For instance, the following expressions may be well-typed:

- $\Lambda X.((\lambda x{:}T.u)\ u)$ because the inner expression is non-expansive. $\Lambda X.(\text{let } x : T = u \text{ in } u)$, which is its syntactic sugar as well,
- $\text{let } x : \forall X.\,\text{list } X = \Lambda X.(t_1\ t_2) \text{ in } t$ because X appears only positively in the type of t_1.

Objective Caml implements both refinements.

In practice

In fact, $\Lambda X.t$ need only be forbidden when X appears in the type of some exposed expansive terms (at some negative occurrence) where exposed subterms are those that do not appear under some λ-abstraction.

For instance, the expression

$$\begin{aligned}&\text{let } x : \forall X.\,\text{int} \times X \to X = \\ &\quad \Lambda X.(\text{ref } (1+2), \lambda x{:}X.\,x) \\ &\text{in } t\end{aligned}$$

may be well-typed because X appears only in the type of non-expansive exposed expressions.

Conclusion

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.

Conclusion

In a type-and-effect system [Lucassen and Gifford, 1988; Talpin and Jouvelot, 1994], or in a type-and-capability system [Charguéraud and Pottier, 2008], the type system indicates which expressions may allocate new references, and at which type.

There, the value restriction is no longer necessary.

However, if one extends a type-and-capability system with a mechanism for *hiding* state, the need for the value restriction re-appears.

Contents

- Why polymorphism?
- Polymorphic λ-calculus
- Type soundness
- Type erasing semantics
- Polymorphism and references
- Damas and Milner's type system

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.

Some intuitions on the definition of ML

The idea behind the definition of ML is to make a small extension of simply-typed λ-calculus that enables to factor out several occurrences of the same subexpression t_1 in a term of the form $[x \mapsto a_1]a_2$ using a let-binding form $\text{let}\, x = a_1 \,\text{in}\, a_2$ so as to avoid code duplication.

Expressions of the simply-typed λ-calculus are extended with as a primitive form of let-binding, which can also be viewed as a way of annotating some redexes $(\lambda x. a_2)\, a_1$ in the source program.

Some intuitions on the definition of ML

This provides 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:

$$\text{let } x = a_1 \text{ in } a_2 \quad \longrightarrow \quad a_1; [x \mapsto a_1] a_2$$

Notice that we introduce a sequence starting with a_1 and not just $[x \mapsto a_1] a_2$.

Why?

Some intuitions on the definition of ML

This provides 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:

$$\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \quad\longrightarrow\quad a_1; [x \mapsto a_1]a_2$$

Notice that we introduce a sequence starting with a_1 and not just $[x \mapsto a_1]a_2$.

This is to enforce well-typedness of a_1 in the pathological case where x does not appear free in a_2.

If we disallow this pathological case (*e.g.* well-formedness could require that x always occurs in a_2) then we could just use the more intuitive rewrite rule:

$$\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \quad\longrightarrow\quad [x \mapsto a_1]a_2$$

Some intuitions on the definition of ML

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

- they have *intrinsic* exponential complexity;
- separate compilation prevents 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

The language ML is normally presented in its implicitly-typed version.

Terms are now given by:

$$a ::= x \mid \lambda x. a \mid a\, a \mid \text{let } x = a \text{ in } a \mid \ldots$$

The *let* construct is no longer sugar for a β-redex: it is now a primitive form, as it will be typed especially.

Types and type schemes

The language of types lies between those for simply-typed λ-calculus and System F; it is stratified between *types* and *type schemes*.

The syntax of *types* is that of simply-typed λ-calculus:

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

A separate category of *type schemes* is introduced:

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

All quantifiers must appear in *prenex position*, so type schemes are less expressive than System F types.

We often write $\forall \vec{X}.T$ as a short hand for $\forall X_1. \ldots \forall X_n.T$.

When viewed as a subset of System F, one must think of *type schemes* are the primary notion of types, of which *types* are a subset.

Typing judgments

An ML typing context Γ binds program variables to *type schemes*.

In the implicitly-typed presentation, type variables are usually introduced implicitly and not part of Γ. However, we keep below the equivalent presentation where type variables are declared in Γ.

Judgments now take the form:

$$\Gamma \vdash a : S$$

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

Typing rules

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

Var
$$\Gamma \vdash x : \Gamma(x)$$

Abs
$$\frac{\Gamma, x : T_0 \vdash a : T}{\Gamma \vdash \lambda x. a : T_0 \to T}$$

App
$$\frac{\Gamma \vdash a_1 : T_2 \to T_1 \qquad \Gamma \vdash a_2 : T_2}{\Gamma \vdash a_1\, a_2 : T_1}$$

Let
$$\frac{\Gamma \vdash a_1 : S \qquad \Gamma, x : S \vdash a_2 : T}{\Gamma \vdash \text{let } x = a_1 \text{ in } a_2 : T}$$

Gen
$$\frac{\Gamma, X \vdash a : S}{\Gamma \vdash a : \forall X. S}$$

Inst
$$\frac{\Gamma \vdash a : \forall X. S}{\Gamma \vdash a : [X \mapsto T]S}$$

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 implicitly-typed System F. Except that *Type variables are instantiated with monotypes.*

Explicitly-typed terms

In proofs, we also use the explicitly-typed version of ML:

$$t ::= x \mid \lambda x{:}T.t \mid t\ t \mid \Lambda X.t \mid t\ T \mid \mathsf{let}\ x : S = t\ \mathsf{in}\ t \ldots$$

The subset of $[F]$ whose type erasure is in ML is defined by:

Var
$$\Gamma \vdash x : \Gamma(x)$$

Abs
$$\frac{\Gamma, x : T_0 \vdash t : T}{\Gamma \vdash \lambda x{:}T_0.t : T_0 \to T}$$

App
$$\frac{\Gamma \vdash t_1 : T_2 \to T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1}$$

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

Tabs
$$\frac{\Gamma, X \vdash t : S}{\Gamma \vdash \Lambda X.t : \forall X.S}$$

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

Example

Here is a simple type derivation that exploits polymorphism:

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

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

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

A non-example

By contrast, this term is *ill-typed:*

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

A non-example

By contrast, 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 f must be assigned a type T that must be simultaneously of the form $\mathsf{int} \rightarrow T_1$ and $\mathsf{bool} \rightarrow T_2$, but there is not such type.

Recall that this term is well-typed in implicitly-typed System F because f can be assigned, for instance, the polymorphic type $\forall X.X \rightarrow X$.

Type soundness for ML

Since ML is a subset of [F], which has been proved sound, we know that ML is sound, *i.e.* that ML programs cannot go wrong.

This also implies that progess holds in ML.

However, we do not know whether subject reduction holds for ML. Indeed, ML expressions could reduce to System *F* expressions that are not in the ML subset.

There are direct proofs of subject reduction for ML. For instance, see Wright and Felleisen [1994], Pottier and Rémy [2005].

We present an indirect proof that reuses subject reduction and progress in System *F*.

Moreover, it uses a syntactic presentation of ML typing rules, which we need anyway.

Type soundness for ML

To verify that subject reduction holds, we proceed as follows:

- We define a syntactic class $\mathcal{R}$ of terms such that for any closed term t in $\mathcal{R}$ and type scheme S, we have

$$\vec{X} \vdash t : S \text{ in ML} \quad \text{iff} \quad \vec{X} \vdash t : S \text{ in } F$$

- We define a judgment $\Gamma \vdash t : S \Rightarrow t'$ that rewrites any explicitly-typed ML term t into a normal term t' in $\mathcal{R}$ preserving the type and the erasure of t.
- We show that reduction in $F \cap \mathcal{R}$ is stable by reduction up to strong ι-normalization.

Normalization of ML terms

Let $\mathcal{R}$ be the set of terms in the following form:

$$\begin{array}{rcl} t \in \mathcal{R} & ::= & \Lambda \vec{X}.s \\ s & ::= & x\,\vec{T} \mid s\;s \mid \lambda x{:}T.\,s \mid \text{let } x : S = t \text{ in } s \end{array}$$

Moreover, we request that the arity of $\vec{T}$ in $x\,\vec{T}$ be the arity of $\vec{X}$ in the type scheme $\forall \vec{X}.T$ assigned to the variable x.

We define a judgment $\Gamma \vdash t : S \Rightarrow t'$, that assuming $\Gamma \vdash t : S$ holds in ML, returns a term $t' \in \mathcal{R}$ with the same type and erasure as t.

Normalization of ML terms

Var

$$\frac{\forall \vec{X}.T = \Gamma(x)}{\Gamma \vdash x : \forall \vec{X}.T \Rightarrow \Lambda \vec{X}.x\ \vec{X}}$$

Tabs

$$\frac{\Gamma, X \vdash t : S \Rightarrow t'}{\Gamma \vdash \Lambda X.t : \forall X.S \Rightarrow \Lambda X.t'}$$

Tapp

$$\frac{\Gamma \vdash t : \forall X.S \Rightarrow \Lambda X.t'}{\Gamma \vdash t\ T : [X \mapsto T]S \Rightarrow [X \mapsto T]t'}$$

Abs

$$\frac{\Gamma, x : T_0 \vdash t : T \Rightarrow s}{\Gamma \vdash \lambda x{:}T_0.t : T_0 \to T \Rightarrow \lambda x{:}T_0.s}$$

App

$$\frac{\Gamma \vdash t_1 : T_2 \to T_1 \Rightarrow s_1 \qquad \Gamma \vdash t_2 : T_2 \Rightarrow s_2}{\Gamma \vdash t_1\ t_2 : T_1 \Rightarrow s_1\ s_2}$$

Let

$$\frac{\Gamma \vdash t_1 : S \Rightarrow t_1' \qquad \Gamma, x : S \vdash t_2 : T \Rightarrow s_2}{\Gamma \vdash \text{let } x : S = t_1 \text{ in } t_2 : T \Rightarrow \text{let } x : S = t_1' \text{ in } s_2}$$

Normalization of ML terms

The normalization performs:

- type η-expansion of every occurrence of a variable according to the arity of its type scheme (Rule Var). This ensures that every occurrence of a type variable will be fully specialized.
- type β-reduction (Rule Tapp). This cancels type applications of type abstractions. As a result. elaborated terms do not contain any type ι-redex.

Normalization verifies the following property:
If $\Gamma \vdash t : S$ in ML, then there exists t' such that

- $\Gamma \vdash t : S \Rightarrow t'$
- t' is in $\mathcal{R}$ if t' and free variables of t' are fully applied (with respect to the arity of their type scheme in Γ)
- $\Gamma \vdash t' : S$ in ML

Back to subject reduction

Observe that β-reduction of expressions preserves membership in $\mathcal{R}$:

$$(\lambda x{:}T.\,t)\;v \longrightarrow [x \mapsto v]t$$

As x is bound to a type, its occurrences in t are not specialized.

However, let-reduction does not preserve membership in $\mathcal{R}$.

$$\text{let } x : \forall\vec{X}.T = v \text{ in } t \longrightarrow [x \mapsto v]t$$

Occurrences of $x\,\vec{T}$ in t become $v\,\vec{T}$ which may contain ι-redexes.

Fortunately, v is necessarily of the form $\Lambda\vec{X}.v'$ where the arity of $\vec{X}$ is equal to that of $\vec{T}$. Hence, we may immediately perform a ι-reduction that brings the term back in $\mathcal{R}$.

However. . . .

Back to subject reduction

However, the ι-reduction step is a strong reduction (usually not in an a call-by-value evaluation context). Fortunately, strong ι-reduction is also type-preserving (easy proof).

In summary, assume that $\vec{X} \vdash a_1 : S$ in ML and $a_1 \longrightarrow a_2$.

- There exists t_1 such that $\vec{X} \vdash t_1 : S$ in ML and $\lceil t_1 \rceil = a_1$.
- By normalization, there exists t_1' in $\mathcal{R}$ such that $\vec{X} \vdash t_1' : S$ in ML and $\lceil t_1' \rceil = a_1$.
- By the type-erasing semantics, there exists t_2 in F whose type erasure is a_2 and such that $t_1 \longrightarrow_\beta t_2$ (since t_1 iss ι-normal).
- By subject reduction, $\vec{X} \vdash t_2 : S$ holds in F.
- By strong type-β-reduction, the type-normal form t_2' of t_2 verifies $\vec{X} \vdash t_2' : S$ in F. and t_2' is in $\mathcal{R}$. Moreover, $\lceil t_2' \rceil$ is $\lceil t_2 \rceil$.
- Therefore, $\vec{X} \vdash a_2 : S$ is in ML.

Syntax-directed presentation of ML

Terms of $\mathcal{R}$ are so constraints that their typing derivations become syntax directed.

More precisely, consider the following set of typing-rules:

Var-Inst
$$\frac{\forall \vec{X}.T = \Gamma(x)}{\Gamma \vdash_{\mathcal{R}} x\,\vec{T} : [\vec{X} \mapsto \vec{T}]T}$$

Abs
$$\frac{\Gamma, x : T_0 \vdash_{\mathcal{R}} t : T}{\Gamma \vdash_{\mathcal{R}} \lambda x{:}T_0.t : T_0 \to T}$$

App
$$\frac{\Gamma \vdash_{\mathcal{R}} t_1 : T_2 \to T_1 \qquad \Gamma \vdash_{\mathcal{R}} t_2 : T_2}{\Gamma \vdash_{\mathcal{R}} t_1\ t_2 : T_1}$$

Let-Gen
$$\frac{\Gamma, \vec{X} \vdash_{\mathcal{R}} t_1 : T_1 \qquad \Gamma, x : \forall \vec{X}.T_1 \vdash_{\mathcal{R}} t_2 : T_2}{\Gamma \vdash_{\mathcal{R}} \mathsf{let}\ x : \forall \vec{X}.T_1 = \Lambda \vec{X}.t_1\ \mathsf{in}\ t_2 : T_2}$$

If t is in $\mathcal{R}$, then the judgments $\Gamma \vdash t : T$ and $\Gamma \vdash_{\mathcal{R}} t : T$ are equivalent.

Syntax-directed presentation for ML Side result

By dropping type information in terms, we obtain a syntax-directed presentation of the typing rules for implicitly-typed terms.

Var-Inst

$$\frac{\forall \vec{X}. T = \Gamma(x)}{\Gamma \vdash_s x : [\vec{X} \mapsto \vec{T}]T}$$

Let-Gen

$$\frac{\Gamma, \vec{X} \vdash_s a_1 : T_1 \qquad \Gamma, x : \forall \vec{X}. T_1 \vdash_s a_2 : T_2}{\Gamma \vdash_s \text{let } x = a_1 \text{ in } a_2 : T_2}$$

Abs

$$\frac{\Gamma, x : T_0 \vdash_s a : T}{\Gamma \vdash_s \lambda x. a : T_0 \to T}$$

App

$$\frac{\Gamma \vdash_s a_1 : T_2 \to T_1 \qquad \Gamma \vdash_s a_2 : T_2}{\Gamma \vdash_s a_1\ a_2 : T_1}$$

The judgments $\Gamma \vdash a : T$ and $\Gamma \vdash_s a : T$ are equivalent.

Syntax-directed presentation for ML Side result

For type inference, we rather use the following equivalent presentation where type variables are not explicitly declared in the typing context:

Var-Inst

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

Let-Gen

$$\frac{\Gamma \vdash a_1 : T_1 \qquad \vec{X} \# \Gamma \qquad \Gamma, x : \forall \vec{X}.\,T_1 \vdash a_2 : T_2}{\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : T_2}$$

Abs

$$\frac{\Gamma, x : T_0 \vdash a : T}{\Gamma \vdash \lambda x.\,a : T_0 \to T}$$

App

$$\frac{\Gamma \vdash a_1 : T_2 \to T_1 \qquad \Gamma \vdash a_2 : T_2}{\Gamma \vdash a_1\ a_2 : T_1}$$

In this system, the substitution lemma can be restated as follows:

Lemma (Substitution lemma)

Typings are stable by substitution.
If $\Gamma \vdash a : T$ then $\varphi\Gamma \vdash a : \varphi T$. for any substitution φ.

Bibliography I

(Most titles have a clickable mark "▷" that links to online versions.)

▷ Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

▷ Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

▷ Jacques Garrigue. Relaxing the value restriction. In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, April 2004.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université Paris 7, June 1972.

Bibliography II

- Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1990.
- Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *ACM SIGPLAN Conference on History of Programming Languages*, June 2007.
- Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of system F. In *ACM International Conference on Functional Programming (ICFP)*, pages 27–38, August 2003.
- Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris 7, June 1992.
- John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–57, January 1988.

Bibliography III

- Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.
- Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–283, January 1996.
- John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2–3):211–249, 1988.
- Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
- Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 41–53, 2001.

Bibliography IV

- Benjamin C. Pierce. Types and Programming Languages. MIT Press, 2002.
- Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.
- Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.
- François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
- John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, April 1974.

Bibliography V

- John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.
- Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000.
- Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 11(2):245–296, 1994.
- Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179 (1):1–18, 2002.
- Philip Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, September 1989.
- Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1–3):201–226, May 2007.

Bibliography VI

- J. B. Wells. The essence of principal typings. In *International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer, 2002.
- J. B. Wells. The undecidability of Mitchell's subtyping relation. Technical Report 95-019, Computer Science Department, Boston University, December 1995.
- J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3): 111–156, 1999.
- Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.
- Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.