MPRI, Typage

Didier Rémy
(With course material from François Pottier)

October 05, 2013

INRIA

Plan of the course

Introduction

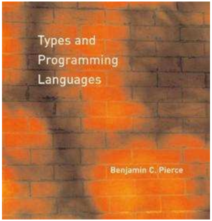
Simply-typed λ-calculus

Polymorphism and System F

Polymorphism and System F

Messages

- Course notes are on the web: `http://gallium.inria.fr/~remy/mpri/`
- Paper course notes and slides and handwritten nodes are allowed for written exams, but books and all electronic devices are forbidden.
- The partial exam on will take place on Tuesday November 03. It will cover the beginning of the course (all lessons covered before the exam). The final exam will cover the *whole* course—not just the end.
- Further reading Pierce [2002]



Excellent book .
Include chapters that cover the two first lessons.

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:

$$\forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha \to \mathit{list}\,\alpha$$

which *instantiates* to the monomorphic types:

$$\begin{gathered}(\mathit{int} \to \mathit{int} \to \mathit{bool}) \to \mathit{list\ int} \to \mathit{list\ int}\\(\mathit{bool} \to \mathit{bool} \to \mathit{bool}) \to \mathit{list\ bool} \to \mathit{list\ bool}\\\ldots\end{gathered}$$

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:

$$(\top \to \top \to \mathit{bool}) \to \mathit{list}\, \top \to \mathit{list}\, \top$$

(The type $\top$ is the type of all values, and the supertype of all types.)

Why isn't this so good?

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:

  $$(\top \to \top \to bool) \to list\ \top \to list\ \top$$

  (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:

$$(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

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

By saying that this term admits the polymorphic type:

$$\forall \alpha_1 \alpha_2.\,(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

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:

$$\forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha \to \mathit{list}\,\alpha$$

then it *knows nothing* about α—it is *parametric* in α—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 α 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 $\forall\alpha.\,\alpha \to \alpha$ has only a few inhabitants, which ones?

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 $\forall\alpha.\,\alpha \to \alpha$ has only *one* inhabitant, up to β-equivalence, namely the identity.

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.

Similarly, the type of the list sorting function

$$\forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha \to \mathit{list}\,\alpha$$

reveals a *“free theorem”* about its behavior!

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.

Similarly, the type of the list sorting function

$$\forall \alpha. (\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha \to \mathit{list}\,\alpha$$

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

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

$$(\forall x, y, cmp\ (f\ x)\ (f\ y) = cmp\ x\ y) \implies \forall \ell, sort\ (map\ f\ \ell) = map\ f\ (sort\ \ell)$$

Note that there are many inhabitants of this type, but they all satisfy this free theorem.

Can you give a few?

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.

Similarly, the type of the list sorting function

$$\forall \alpha. (\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\, \alpha \to \mathit{list}\, \alpha$$

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

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

$$(\forall x, y, \mathit{cmp}\ (f\ x)\ (f\ y) = \mathit{cmp}\ x\ y) \implies \forall \ell, \mathit{sort}\ (\mathit{map}\ f\ \ell) = \mathit{map}\ f\ (\mathit{sort}\ \ell)$$

Note that there are many inhabitants of this type, but they all satisfy this free theorem. (e.g. a function that sorts in reverse order, or a function that removes duplicates)

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.

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].

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.

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].

Unfortunately, parametricity theorems are invalidated or degenerate in the presence of side effects

?

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.

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].

Unfortunately, parametricity theorems are invalidated or degenerate in the presence of side effects (non-termination, exceptions, or references).

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.

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].

Unfortunately, parametricity theorems are invalidated or degenerate in the presence of side effects (non-termination, exceptions, or references).

While most programs use side effects and side effects cannot be ignored when reasoning globally, many parts of programs do not use them and reasoning locally as if they where no side effects is still often helpful.

Parametricity plays an important role in the study of functional programming languages and remains a guideline when programming.

Ad hoc versus parametric

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

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

$$\begin{gathered} \mathit{int} \to \mathit{int}, \\ \mathit{bool} \to \mathit{bool}, \\ \ldots \\ \forall \alpha. \alpha \to \alpha \end{gathered}$$

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

$$\begin{gathered} \mathit{int} \to \mathit{int} \to \mathit{int}, \\ \mathit{float} \to \mathit{float} \to \mathit{float}, \\ \ldots \\ \textit{but not} \\ \forall \alpha. \alpha \to \alpha \to \alpha \end{gathered}$$

Ad hoc versus parametric: second 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. $\lambda x.\, show\;(read\;x)$).

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:

$$\forall \alpha.\, \mathit{Num}\, \alpha \Rightarrow \alpha \to \alpha \to \alpha$$

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

In this course, we are mostly interested only in the simplest form of parametric polymorphism.

Still, we will study Haskell type classes and show how the two views can be reconciled in some cases.

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 with universal quantification:

$$\tau ::= \ldots \mid \forall\alpha.\tau$$

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:

$$\textsc{Tabs}\quad \frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\tau} \qquad\qquad \textsc{Tapp}\quad \frac{\Gamma \vdash M : \forall\alpha.\tau}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}$$

Terms are extended accordingly:

$$M ::= \ldots \mid \Lambda\alpha.M \mid M\ \tau$$

Type variables are explicitly bound and appear in type environments.

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

Well-formedness of environment

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

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

However, a looser style would also be possible.

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

The stricter presentation is easier to manipulate in proofs;
it is also easier to mechanize.

Well-formedness of environments and types

Well-formedness of environments, written $\vdash \Gamma$ and well-formedness of types, written $\Gamma \vdash \tau$, may also be defined *recursively* by inference rules:

$$\textsc{WfEnv-Empty}\quad \vdash \varnothing$$

$$\textsc{WfEnvVar}\quad \frac{\vdash \Gamma \qquad x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau}$$

$$\textsc{WfEnvTvar}\quad \frac{\vdash \Gamma \qquad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha}$$

$$\textsc{WfTypeVar}\quad \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha}$$

$$\textsc{WfTypeArrow}\quad \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau}$$

$$\textsc{WfTypeForall}\quad \frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha.\, \tau}$$

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 appears in the conclusion does not appear in any premise.

$$\textsc{Var}\quad \frac{x : \tau \in \Gamma \qquad \boxed{\vdash \Gamma}}{\Gamma \vdash x : \tau} \qquad\qquad \textsc{Tapp}\quad \frac{\Gamma \vdash M : \forall \alpha.\tau \qquad \boxed{\Gamma \vdash \tau'}}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}$$

Explicit well-formedness is more precise and better suited for mechanized proofs. Explicit well-formedness is recommended.

However, we choose to leave well-formedness conditions implicit in this course, as it is a bit verbose and sometimes distracting. Still, we remind implicit well-formedness premises in the definition of typing rules.

Type-passing semantics

We need the following reduction for type-level expressions:

$$(\Lambda\alpha.M)\,\tau \longrightarrow [\alpha \mapsto \tau]M \qquad (\iota)$$

Then, there is a choice.

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

?

Type-passing semantics

We need the following reduction for type-level expressions:

$$(\Lambda\alpha.M)\,\tau \longrightarrow [\alpha \mapsto \tau]M \qquad (\iota)$$

Then, there is a choice.

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

$$V ::= \ldots \mid \Lambda\alpha.M \qquad\qquad E ::= \ldots \mid [\,]\;\tau$$

However,

?

Type-passing semantics

We need the following reduction for type-level expressions:

$$(\Lambda\alpha.M)\ \tau \longrightarrow [\alpha \mapsto \tau]M \qquad (\iota)$$

Then, there is a choice.

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

$$V ::= \ldots \mid \Lambda\alpha.M \qquad\qquad E ::= \ldots \mid [\,]\ \tau$$

However, this defines a type-passing semantics!

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

Type-erasing semantics

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

?

Type-erasing semantics

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

$$V ::= \ldots \mid \Lambda\alpha.V \qquad\qquad E ::= \ldots \mid [\,]\ \tau \mid \Lambda\alpha.[\,]$$

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

?

Type-erasing semantics

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

$$V ::= \ldots \mid \Lambda\alpha.V \qquad\qquad E ::= \ldots \mid [\,]\;\tau \mid \Lambda\alpha.[\,]$$

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

$$(\Lambda\alpha.V)\;\tau \longrightarrow [\alpha \mapsto \tau]V \qquad (\iota)$$

We now have:

$$\Lambda\alpha.((\lambda y : \alpha.\, y)\; V) \longrightarrow \Lambda\alpha.V$$

We verify below that this defines a type-erasing semantics, indeed.

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.
- it blocks reduction of polymorphic expressions: *e.g.*, if f is list flattening of type $\forall\alpha.\, \mathit{list}\,(\mathit{list}\,\alpha) \to \mathit{list}\,\alpha$, the monomorphic function $(f\ \mathit{int}) \circ (f\ (\mathit{list}\,\mathit{int}))$ reduces to $\Lambda x.f\ (f\ x)$, while its more general polymorphic version $\Lambda\alpha.(f\ \alpha) \circ (f\ (\mathit{list}\,\alpha))$ is irreducible.
- 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, with the semantics of most programming languages. It also exhibits difficulties when adding side effects while the type-passing semantics does not.

In the following, we choose a type-erasing semantics.

Notice that we allow evaluation under a type abstraction as a consequence of choosing a type-erasing semantics—and not the converse.

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 $\Lambda\alpha.M$ when M is a value-form, then the type-passing and type-erasing semantics coincide.

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

This restriction will be chosen 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.

Once, we have verified that the semantics is indeed type-preserving, many properties can be *transferred 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:

$$\begin{aligned} pair &\triangleq \Lambda\alpha_1.\Lambda\alpha_2.\lambda x_1{:}\alpha_1.\,\lambda x_2{:}\alpha_2.\,\Lambda\beta.\lambda y{:}\alpha_1 \to \alpha_2 \to \beta.\, y\; x_1\; x_2 \\ proj_i &\triangleq \Lambda\alpha_1.\Lambda\alpha_2.\lambda y{:}\forall\beta.\,(\alpha_1 \to \alpha_2 \to \beta) \to \beta.\, y\; \alpha_i\; (\lambda x_1{:}\alpha_1.\,\lambda x_2{:}\alpha_2.\, x_i) \end{aligned}$$

$$\begin{aligned} \lceil pair \rceil &\triangleq \lambda x_1.\,\lambda x_2.\,\lambda y.\, y\; x_1\; x_2 \\ \lceil proj_i \rceil &\triangleq \lambda y.\, y\; (\lambda x_1.\,\lambda x_2.\, x_i) \end{aligned}$$

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 expressiveness 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.

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

Let us first illustrate an extension of System F with primitive pairs.
(We will them generalize it to arbitrary constructors and destructors.)

Constructors and destructors Pairs

Types are extended with a type constructor × of arity 2:

$$\tau ::= \ldots \mid \tau \times \tau$$

Expressions are extended with a constructor $(\cdot, \cdot)$ and two destructors *proj*$_1$ and *proj*$_2$ with the respective signatures:

$$\begin{array}{rl} Pair : & \forall \alpha_1. \forall \alpha_2. \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2 \\ proj_i : & \forall \alpha_1. \forall \alpha_2. \alpha_1 \times \alpha_2 \to \alpha_i \end{array}$$

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 (all cases but one). Hence, values are extended as follows:

$$\begin{array}{rcl} V & ::= & \ldots \mid Pair \mid Pair\ \tau \mid Pair\ \tau\ \tau \mid Pair\ \tau\ \tau\ V \mid Pair\ \tau\ \tau\ V\ V \\ & & \mid proj_i \mid proj_i\ \tau \mid proj_i\ \tau\ \tau \end{array}$$

Constructors and destructors Pairs

We add the two following reduction rules:

$$proj_i\ \tau_1\ \tau_2\ (pair\ \tau_1'\ \tau_2'\ V_1\ V_2) \longrightarrow V_i \qquad (\delta_{pair})$$

Comments?

Constructors and destructors Pairs

We add the two following reduction rules:

$$proj_i\ \tau_1\ \tau_2\ (pair\ \tau_1'\ \tau_2'\ V_1\ V_2) \longrightarrow V_i \qquad (\delta_{pair})$$

Comments?

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

  Instead, one could have defined the rule:

  $$proj_i\ \tau_1\ \tau_2\ (pair\ \tau_1\ \tau_2\ V_1\ V_2) \longrightarrow V_i \qquad (\delta_{pair}')$$

  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.

  Interestingly, with δ_{pair}', the proof obligation is simpler in subject reduction but replaced by a stronger proof obligation in progress.

Constructors and destructors

Pairs

We add the two following reduction rules:

$$proj_i\ \tau_1\ \tau_2\ (pair\ \tau_1'\ \tau_2'\ V_1\ V_2) \longrightarrow V_i \qquad (\delta_{pair})$$

Comments?

- This presentation forces the programmer to specify the types of the components of the pair.

  However, since this is an explicitly type presentation, these types are already known from the arguments of the pair (when present)

  This 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 *G*, with their arity *arity* (*G*). We assume that types respect the arities of type constructors.

A type $G\,(\vec{\tau})$ is called a *G*-type.
A *datatype* is a *G*-type for some type constructor *G*.

Let Δ be an initial environment binding constants c of arity n (split into constructors C or destructors d) to signatures of the form:

$$c : \forall \alpha_1. \ldots \forall \alpha_k. \underbrace{\tau_1 \to \ldots \tau_n}_{\textit{arity}(c)} \to \tau$$

We require that

- τ be is a datatype whenever c is a constructor (for progress);
- n is strictly positive when c is a destructor
  (nullary destructors introduce pathological cases for little benefit).

Constructors and destructors General case

Expressions are extended with constants: Constants are typed as variables, but their types are looked up in the initial environment Δ:

$$M ::= \ldots \mid c \qquad\qquad \frac{c : \tau \in \Delta}{\Gamma \vdash c : \tau}\,\textsc{Cst}$$

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

$$\begin{array}{rcll} V & ::= & \ldots & \\ & \mid & C\;\tau_1\;\ldots\;\tau_p\;V_1\;\ldots\;V_q & q \le \mathit{arity}\,(C) \\ & \mid & d\;\tau_1\;\ldots\;\tau_p\;V_1\;\ldots\;V_q & q < \mathit{arity}\,(d) \end{array}$$

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

$$d\;\tau_1\;\ldots\;\tau_k\;V_1\;\ldots\;V_n \longrightarrow M \qquad (\delta_d)$$

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 $\vec{\alpha} \vdash M_1 : \tau$ and $M_1 \longrightarrow_\delta M_2$ then $\vec{\alpha} \vdash M_2 : \tau$.

Progress for constants: If $\vec{\alpha} \vdash M_1 : \tau$ and M_1 is of the form $d\ \tau_1\ \ldots\ \tau_k\ V_1\ \ldots\ V_n$ where $n = \mathit{arity}\,(d)$, then there exists M_2 such that $M_1 \longrightarrow M_2$.

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

$$\mathit{fix} : \forall \alpha. \forall \beta. ((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta \qquad \in \Delta$$

of arity 2, together with the δ-rule

$$\mathit{fix}\ \tau_1\ \tau_2\ V_1\ V_2 \longrightarrow V_1\ (\mathit{fix}\ \tau_1\ \tau_2\ V_1)\ V_2 \qquad (\delta_{\mathit{fix}})$$

It is straightforward to check the assumptions:

?

Example Fixpoint

We introduce a destructor

$$\mathit{fix} : \forall\alpha.\,\forall\beta.\,((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta \qquad \in \Delta$$

of arity 2, together with the δ-rule

$$\mathit{fix}\;\tau_1\;\tau_2\;V_1\;V_2 \longrightarrow V_1\;(\mathit{fix}\;\tau_1\;\tau_2\;V_1)\;V_2 \qquad (\delta_{\mathit{fix}})$$

It is straightforward to check the assumptions:

- Progress is obvious,

?

Example Fixpoint

We introduce a destructor

$$\mathit{fix} : \forall\alpha.\,\forall\beta.\,((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta \qquad \in \Delta$$

of arity 2, together with the δ-rule

$$\mathit{fix}\;\tau_1\;\tau_2\;V_1\;V_2 \longrightarrow V_1\;(\mathit{fix}\;\tau_1\;\tau_2\;V_1)\;V_2 \qquad (\delta_{\mathit{fix}})$$

It is straightforward to check the assumptions:

- Progress is obvious, since δ_{fix} works for any values V_1 and V_2.

Example

Fixpoint

We introduce a destructor

$$\mathit{fix} : \forall\alpha.\,\forall\beta.\,((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta \qquad \in \Delta$$

of arity 2, together with the δ-rule

$$\mathit{fix}\; \tau_1\; \tau_2\; V_1\; V_2 \longrightarrow V_1\; (\mathit{fix}\; \tau_1\; \tau_2\; V_1)\; V_2 \qquad (\delta_{\mathit{fix}})$$

It is straightforward to check the assumptions:

- Progress is obvious, since δ_{fix} works for any values V_1 and V_2.
- Subject reduction is also straightforward.

?

Example Fixpoint

We introduce a destructor

$$\mathit{fix} : \forall\alpha.\,\forall\beta.\,((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta \qquad \in \Delta$$

of arity 2, together with the δ-rule

$$\mathit{fix}\; \tau_1\; \tau_2\; V_1\; V_2 \longrightarrow V_1\; (\mathit{fix}\; \tau_1\; \tau_2\; V_1)\; V_2 \qquad (\delta_{\mathit{fix}})$$

It is straightforward to check the assumptions:

- Progress is obvious, since δ_{fix} works for any values V_1 and V_2.
- Subject reduction is also straightforward.
  Assume that $\Gamma \vdash \mathit{fix}\; \tau_1\; \tau_2\; V_1\; V_2 : \tau$. By inversion of typing rules, τ must be equal to τ_2, V_1 and V_2 must be of types $(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$ and τ_1 in the typing context Γ. We may then easily build a derivation of the judgment $\Gamma \vdash V_1\; (\mathit{fix}\; \tau_1\; \tau_2\; V_1)\; V_2 : \tau$

Exercise Lists

1) Formulate the extension of System F with lists as constants.

2) Check that this extension is sound.

Exercise

Lists

1) Formulate the extension of System F with lists as constants.

2) Check that this extension is sound.

Solution

1) We introduce a new unary type constructor *list* ; two constructors *Nil* · and *Cons* of types $\forall\alpha.\,\mathit{list}\,\alpha$ and $\forall\alpha.\,\alpha\to\mathit{list}\,\alpha\to\mathit{list}\,\alpha$; and one destructor *matchlist* · · ·· of type:

$$\forall\alpha\beta.\,\mathit{list}\,\alpha\to\beta\to(\alpha\to\mathit{list}\,\alpha\to\beta)\to\beta$$

with the two reduction rules:

$$\begin{array}{l}\mathit{matchlist}\,M\,(\mathit{Nil}\,M')\,V_n\,V_c\longrightarrow V_n\\ \mathit{matchlist}\,M\,(\mathit{Cons}\,M'\,V_h\,V_t)\,V_n\,V_c\longrightarrow V_c\,V_h\,V_t\end{array}$$

2) See the case of pairs in the course.

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:

- inversion of typing judgments
- permutation and weakening
- expression substitution
- type substitution
- compositionality

Type soundness

Weakening

Lemma (Weakening)

Assume $\Gamma \vdash M : \tau$.
1) If $x \# \Gamma$, *then* $\Gamma, x : \tau' \vdash M : \tau$.
2) If $\beta \# \Gamma$, *then* $\Gamma, \beta \vdash M : \tau$.

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 M, then by cases on M 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 $\Gamma, \Delta, \Delta', \Gamma' \vdash M : \tau$ *and* $\Delta \# \Delta'$, *then* $\Gamma, \Delta', \Delta, \Gamma' \vdash M : \tau$.

Type Soundness

Type substitution

Lemma (Expression substitution, strengthened)

If $\Gamma, x : \tau_0, \Gamma' \vdash M : \tau$ *and* $\Gamma \vdash M_0 : \tau_0$ *then* $\Gamma, \Gamma' \vdash [x \mapsto M_0]M : \tau$.

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 M, as this does not complicate the proof.

Type Soundness Type substitution

Lemma (Type substitition , strengthened)

If $\Gamma, \alpha, \Gamma' \vdash M : \tau'$ *and* $\Gamma \vdash \tau$ *then* $\Gamma, [\alpha \mapsto \tau]\Gamma' \vdash [\alpha \mapsto \tau]M : [\alpha \mapsto \tau]\tau'$.

The proof is by induction on M.

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 complicate the proof, but yields a stronger result.

Compositionality

Lemma (Compositionality)

If $\Gamma \vdash E[M] : \tau$*, then there exists* $\bar{\alpha}$ *and* τ' *such that* $\Gamma, \bar{\alpha} \vdash M : \tau'$ *and all* M' *verifying* $\Gamma, \bar{\alpha} \vdash M' : \tau'$ *also verify* $\Gamma \vdash E[M'] : \tau$*.*

Extension of Γ by variables is needed because evaluation proceeds under type abstractions.

Type Soundness Subject reduction

Proof of subject reduction.

The proof is by induction on M.

Using the previous lemmas it is straightforward.

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

Type soundness Progress

Progress is restated as follows:

Theorem (Progress, strengthened)

A well-typed, irreducible closed term is a value:
if $\vec{\alpha} \vdash M : \tau$ *and* $M \not\longrightarrow$ *, then* M *is some value* V*.*

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

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

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 $\bar{\alpha} \vdash V : \tau$

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

The last case can be refined by partitioning constructors into their associated type-constructor: If τ is a G-constructed type, then V is a value constructed with a G-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 more 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]).

See also the 2011-2012 partial exam.

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 an explicitly-typed term.

We use letters a, v, and e 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. Then, there are two new rules :

$$\textsc{if-Tabs}\quad \frac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall \alpha.\tau} \qquad\qquad \textsc{if-Tapp}\quad \frac{\Gamma \vdash a : \forall \alpha.\tau}{\Gamma \vdash a : [\alpha \mapsto \tau_0]\tau}$$

Notice that these rules are not syntax directed.

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

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

In implicitly-typed System F, we could also omit type declarations from the typing environment. (Although, in some extensions of System F, type variables may carry a kind or a bound and must be explicitly introduced.)

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

$$\textsc{Tabs-Bis}\quad \frac{\Gamma \vdash a : \tau \qquad \alpha \# \Gamma}{\Gamma \vdash a : \forall \alpha . \tau}$$

Why is the side condition important?...

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

Omitting the side condition leads to *unsoundness:*

$$\textsc{Tabs-Bis}\;\dfrac{\textsc{Abs}\;\dfrac{\textsc{Tapp}\;\dfrac{\textsc{Broken Tabs}\;\dfrac{\textsc{Var}\;\dfrac{}{x : \alpha_1 \vdash x : \alpha_1}}{x : \alpha_1 \vdash x : \forall \alpha_1 . \alpha_1}}{x : \alpha_1 \vdash x : \alpha_2}}{\varnothing \vdash \lambda x . x : \alpha_1 \to \alpha_2}}{\varnothing \vdash \lambda x . x : \forall \alpha_1 . \forall \alpha_2 . \alpha_1 \to \alpha_2}$$

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

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

Omitting the side condition leads to *unsoundness:*

$$\textsc{Tabs-Bis}\;\dfrac{\textsc{Abs}\;\dfrac{\textsc{Tapp}\;\dfrac{\textsc{Broken Tabs}\;\dfrac{\textsc{Var}\;\dfrac{}{x : \alpha_1 \vdash x : \alpha_1}}{x : \alpha_1 \vdash x : \forall \alpha_1 . \alpha_1}}{x : \alpha_1 \vdash x : \alpha_2}}{\varnothing \vdash \lambda x . x : \alpha_1 \to \alpha_2}}{\varnothing \vdash \lambda x . x : \forall \alpha_1 . \forall \alpha_2 . \alpha_1 \to \alpha_2}$$

$\alpha_1 \in \mathrm{ftv}(x : \alpha_1)$

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

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

This is equivalent to using an ill-formed typing environment :

$$\textsc{Tabs}\;\dfrac{\textsc{Abs}\;\dfrac{\textsc{Tapp}\;\dfrac{\textsc{Broken Tabs}\;\dfrac{\textsc{Broken Var}\;\dfrac{}{x : \alpha_1, \alpha_1 \vdash x : \alpha_1}}{x : \alpha_1 \vdash x : \forall \alpha_1.\alpha_1}}{x : \alpha_1 \vdash x : \alpha_2}}{\varnothing \vdash \lambda x{:}\alpha_1.\, x : \alpha_1 \to \alpha_2}}{\varnothing \vdash \Lambda\alpha_1.\Lambda\alpha_2.\lambda\alpha_1{:}x.\, x : \forall\alpha_1.\forall\alpha_2.\alpha_1 \to \alpha_2}$$

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

This is equivalent to using an ill-formed typing environment :

$$\textsc{Tabs}\ \dfrac{\textsc{Abs}\ \dfrac{\textsc{Tapp}\ \dfrac{\textsc{Broken Tabs}\ \dfrac{\textsc{Broken Var}\ \dfrac{}{x:\alpha_1,\alpha_1 \vdash x:\alpha_1}}{x:\alpha_1 \vdash x:\forall\alpha_1.\alpha_1}}{x:\alpha_1 \vdash x:\alpha_2}}{\varnothing \vdash \lambda x{:}\alpha_1.\,x:\alpha_1 \to \alpha_2}}{\varnothing \vdash \Lambda\alpha_1.\Lambda\alpha_2.\lambda\alpha_1{:}x.\,x:\forall\alpha_1.\forall\alpha_2.\alpha_1 \to \alpha_2}$$

$x:\alpha_1,\alpha_1$ ill-formed

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

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

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

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

Type-erasing typechecking

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

Lemma

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

Trivial.

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

One could write judgements of the form $M \Rightarrow \Gamma \vdash a : \tau$ to mean that the *explicitly tuped* term M witnesses that the *implicitly typed* term a has type τ in the environment Γ.

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:

$$\Lambda\alpha_1.\Lambda\alpha_2.\lambda f : \alpha_1 \to \alpha_2.\, \lambda x : \alpha_1.\, \lambda y : \alpha_1.\,(f\ x, f\ y)$$

This term admits the polymorphic type:

$$\forall\alpha_1.\forall\alpha_2.(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

Quite unsurprising, right?

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:

$$\Lambda\alpha_1.\Lambda\alpha_2.\lambda f : \alpha_1 \to \alpha_2.\,\lambda x : \alpha_1.\,\lambda y : \alpha_1.\,(f\ x, f\ y)$$

This term admits the polymorphic type:

$$\forall\alpha_1.\forall\alpha_2.(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

Quite unsurprising, right? Perhaps more surprising is the fact that this untyped term can be decorated in a different way:

?

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

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

$$\Lambda \alpha_1 . \Lambda \alpha_2 . \lambda f : \alpha_1 \to \alpha_2 . \lambda x : \alpha_1 . \lambda y : \alpha_1 . (f\ x, f\ y)$$

This term admits the polymorphic type:

$$\forall \alpha_1 . \forall \alpha_2 . (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

Quite unsurprising, right? Perhaps more surprising is the fact that this untyped term can be decorated in a different way:

$$\Lambda \alpha_1 . \Lambda \alpha_2 . \lambda f : \forall \alpha . \alpha \to \alpha . \lambda x : \alpha_1 . \lambda y : \alpha_2 . (f\ \alpha_1\ x, f\ \alpha_2\ y)$$

This term admits the polymorphic type:

$$\forall \alpha_1 . \forall \alpha_2 . (\forall \alpha . \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

This begs the question: ...

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

Which of the two is more general?

$$\forall \alpha_1.\forall \alpha_2.(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$
$$\forall \alpha_1.\forall \alpha_2.(\forall \alpha.\, \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

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

Which of the two is more general?

$$\forall\alpha_1.\forall\alpha_2.(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$
$$\forall\alpha_1.\forall\alpha_2.(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

The first one requires x and y to admit a common type, while the second one requires f to be polymorphic.

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

Which of the two is more general?

$$\begin{gathered}\forall\alpha_1.\forall\alpha_2.(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2 \\ \forall\alpha_1.\forall\alpha_2.(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2\end{gathered}$$

The first one requires x and y to admit a common type, while the second one requires f to be polymorphic. *Neither one is 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 $\lceil F \rceil$

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 τ_2 to be an *instance* of τ_1—or, equivalently, for τ_1 to be *more general* than τ_2.

Several definitions are possible...

Syntactic notions of instance in $\lceil F \rceil$

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

$$\textsc{Inst-Gen}\quad \frac{\bar\beta \mathrel{\#} \forall\bar\alpha.\tau}{\forall\bar\alpha.\tau \le \forall\bar\beta.[\vec\alpha \mapsto \vec\tau]\tau}$$

One can show that, if $\tau_1 \le \tau_2$, then any term that has type τ_1 also has type τ_2; that is, the following rule is *admissible*:

$$\textsc{Sub}\quad \frac{\Gamma \vdash a : \tau_1 \qquad \tau_1 \le \tau_2}{\Gamma \vdash a : \tau_2}$$

Perhaps surprisingly, the rule is *not derivable* in our presentation of System F as the proof of admissibility requires weakening.
(It would be derivable if we had left type variables implicit in contexts.)

Syntactic notions of instance in F

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

Assume $\Gamma \vdash M : \tau_1$ and $\tau_1 \leq \tau_2$. How can we see M with type τ_2?

Well, M_1 and M_2 must be of the form $\forall \bar\alpha.\,\tau$ and $\forall \bar\beta.\,[\vec\alpha \mapsto \vec\tau]\tau$ where $\bar\beta \# \forall \bar\alpha.\,\tau$. *W.l.o.g*, we may assume that $\bar\beta \# \Gamma$.

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

$$\textsc{Tabs}^*\;\dfrac{\textsc{Tapp}^*\;\dfrac{\textsc{Weak.}\;\dfrac{\Gamma \vdash M : \forall \vec\alpha.\,\tau \qquad \bar\beta \# \Gamma}{\Gamma, \vec\beta \vdash M : \forall \vec\alpha.\,\tau}}{\Gamma, \vec\beta \vdash M\;\vec\tau : [\vec\alpha \mapsto \vec\tau]\tau}}{\Gamma \vdash \Lambda\vec\beta.M\;\vec\tau : \forall\vec\beta.\,[\vec\alpha \mapsto \vec\tau]\tau}$$

Syntactic notions of instance in F

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

Assume $\Gamma \vdash M : \tau_1$ and $\tau_1 \leq \tau_2$. How can we see M with type τ_2?

Well, M_1 and M_2 must be of the form $\forall \bar\alpha.\, \tau$ and $\forall \bar\beta.\, [\vec\alpha \mapsto \vec\tau]\tau$ where $\bar\beta \# \forall \bar\alpha.\, \tau$. *W.l.o.g*, we may assume that $\bar\beta \# \Gamma$.

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

$$\left.\textsc{Tabs}^*\;\dfrac{\textsc{Tapp}^*\;\dfrac{\textsc{Weak.}\;\dfrac{\Gamma \vdash M : \forall \vec\alpha.\, \tau \qquad \bar\beta \# \Gamma}{\Gamma, \vec\beta \vdash M : \forall \vec\alpha.\, \tau}}{\Gamma, \vec\beta \vdash M\ \vec\tau : [\vec\alpha \mapsto \vec\tau]\tau}}{\Gamma \vdash \Lambda\vec\beta.M\ \vec\tau : \forall \vec\beta.\, [\vec\alpha \mapsto \vec\tau]\tau}\right\}$$

Admissible rule:

$$\textsc{Sub}\;\dfrac{\begin{array}{c}\bar\beta \# \forall \vec\alpha.\, \tau \\ \Gamma \vdash M : \forall \vec\alpha.\, \tau\end{array}}{\Gamma \vdash \Lambda\vec\beta.M\ \vec\tau : \forall \vec\beta.\, [\vec\alpha \mapsto \vec\tau]\tau}$$

Syntactic notions of instance in F

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

Assume $\Gamma \vdash M : \tau_1$ and $\tau_1 \leq \tau_2$. How can we see M with type τ_2?

Well, M_1 and M_2 must be of the form $\forall \bar\alpha.\,\tau$ and $\forall \bar\beta.\,[\vec\alpha \mapsto \vec\tau]\tau$ where $\bar\beta \# \forall \bar\alpha.\,\tau$. *W.l.o.g*, we may assume that $\bar\beta \# \Gamma$.

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

$$\textsc{Tabs}^*\ \dfrac{\textsc{Tapp}^*\ \dfrac{\textsc{Weak.}\ \dfrac{\Gamma \vdash M : \forall\vec\alpha.\,\tau \qquad \bar\beta \# \Gamma\ (1)}{\Gamma, \vec\beta \vdash M : \forall\vec\alpha.\,\tau}}{\Gamma, \vec\beta \vdash M\ \vec\tau : [\vec\alpha \mapsto \vec\tau]\tau}}{\Gamma \vdash \Lambda\vec\beta.M\ \vec\tau : \forall\vec\beta.\,[\vec\alpha \mapsto \vec\tau]\tau}$$

Admissible rule:

$$\textsc{Sub}\ \dfrac{\begin{array}{c}\bar\beta \# \forall\vec\alpha.\,\tau\ (2)\\ \Gamma \vdash M : \forall\vec\alpha.\,\tau\end{array}}{\Gamma \vdash \Lambda\vec\beta.M\ \vec\tau : \forall\vec\beta.\,[\vec\alpha \mapsto \vec\tau]\tau}$$

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

Retyping contexts in F

In F, subtyping is a judgment $\Gamma \vdash \tau_1 \leq \tau_2$ 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.

$$\mathcal{R} ::= [\,] \mid \Lambda\alpha.\mathcal{R} \mid \mathcal{R}\ \tau$$

(Notice that $\mathcal{R}$ are arbitrarily deep, as opposed to evaluation contexts.)

Let us write $\Gamma \vdash \mathcal{R}[\tau_1] : \tau_2$ iff $\Gamma, x : \tau_1 \vdash \mathcal{R}[x] : \tau_2$.

If $\Gamma \vdash M : \tau_1$ and $\Gamma \vdash \mathcal{R}[\tau_1] : \tau_2$, then $\Gamma \vdash \mathcal{R}[M] : \tau_2$,

Then $\Gamma \vdash \tau_1 \leq \tau_2$ iff $\Gamma \vdash \mathcal{R}[\tau_1] : \tau_2$. for some retyping context $\mathcal{R}$.

In System F, retyping contexts can only change *toplevel* polymorphism: they cannot operate under arrow types to weaken the return type of or strengthen the domain of functions.

Another syntactic notion of instance: F_η

Mitchell [1988] defined F_η, a version of $\lceil F \rceil$ extended with a richer *instance* relation as:

Inst-Gen

$$\frac{\bar\beta \# \forall\bar\alpha.\tau}{\forall\bar\alpha.\tau \le \forall\bar\beta.[\vec\alpha \mapsto \vec\tau]\tau}$$

Distributivity

$$\forall\alpha.\,(\tau_1 \to \tau_2) \le (\forall\alpha.\,\tau_1) \to (\forall\alpha.\,\tau_2)$$

Congruence-→

$$\frac{\tau_2 \le \tau_1 \qquad \tau_1' \le \tau_2'}{\tau_1 \to \tau_1' \le \tau_2 \to \tau_2'}$$

Congruence-∀

$$\frac{\tau_1 \le \tau_2}{\forall\alpha.\tau_1 \le \forall\alpha.\tau_2}$$

Transitivity

$$\frac{\tau_1 \le \tau_2 \qquad \tau_2 \le \tau_3}{\tau_1 \le \tau_3}$$

In F_η, Rule Sub must be primitive as it is not admissible (but still sound).

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

Why is a rich notion of instance potentially interesting?

?

Another syntactic notion of instance: F_η

Mitchell [1988] defined F_η, a version of $\lceil F \rceil$ extended with a richer *instance* relation as:

Inst-Gen
$$\frac{\bar\beta \# \forall\bar\alpha.\tau}{\forall\bar\alpha.\tau \le \forall\bar\beta.[\vec\alpha \mapsto \vec\tau]\tau}$$

Distributivity
$$\forall\alpha.\,(\tau_1 \to \tau_2) \le (\forall\alpha.\,\tau_1) \to (\forall\alpha.\,\tau_2)$$

Congruence-→
$$\frac{\tau_2 \le \tau_1 \qquad \tau_1' \le \tau_2'}{\tau_1 \to \tau_1' \le \tau_2 \to \tau_2'}$$

Congruence-∀
$$\frac{\tau_1 \le \tau_2}{\forall\alpha.\tau_1 \le \forall\alpha.\tau_2}$$

Transitivity
$$\frac{\tau_1 \le \tau_2 \qquad \tau_2 \le \tau_3}{\tau_1 \le \tau_3}$$

In F_η, Rule Sub must be primitive as it is not admissible (but still sound).

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

Why is a rich notion of instance potentially interesting?

- More polymorphism.
- More hope of principal types.

A definition of principal typings

A typing of an expression M is a pair Γ, τ such that $\Gamma \vdash M : \tau$.

Ideally, a type system should have *principal typings* [Wells, 2002]:

> *Every well-typed term M admits a principal typing – one whose instances are exactly the typings of M.*

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 (for typings) 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 F_η have principal typings.*

It was shown earlier that *F_η's instance relation is undecidable* [Wells, 1995; Tiuryn and Urzyczyn, 2002] and that *type inference for both System F and 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 M is there a type τ for M in Γ such that all other types of M in Γ are instances of τ.

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 pull back 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 $M \in F$ whose erasure is a and since M 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 τ that reduces to a_2, can we find a term $M_1 \in F$ whose erasure is a_1 and show that M_1 reduces to a term M_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 terms, hence the two reductions cannot coincide *exactly*.

The way to formalize this is to split reduction steps into $\beta\delta$-steps corresponding to β or δ rules that are preserved by type-erasure, and ι-steps corresponding to the reduction of type applications that disappear during 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 terms, hence the two reductions cannot coincide *exactly*.

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

$$\begin{tikzcd}
M_0 \arrow[rr, dashed, "*", "\iota"'] \arrow[dr, dotted] & & M_0' \arrow[r, "\beta\delta"'] \arrow[dl, dotted] & M_1 \arrow[dr, dotted] & \\
& a_0 \arrow[rrr, "\beta\delta"'] & & & a_1
\end{tikzcd}$$

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 terms, hence the two reductions cannot coincide *exactly*.

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

$$\begin{tikzcd}
M_0 \arrow[r, dashed, "*", "\iota"'] \arrow[d, dotted] & M'_0 \arrow[r, "\beta\delta"'] \arrow[dl, dotted] & M_1 \arrow[d, dotted] & & M_j \arrow[r, dashed, "*", "\iota"'] \arrow[d, dotted] & M'_j \arrow[r, "\beta\delta"'] \arrow[dl, dotted] & M_{j+1} \arrow[d, dotted] \\
a_0 \arrow[rr, "\beta\delta"'] & & a_1 & \ldots & a_j \arrow[rr, "\beta\delta"'] & & a_{j+1}
\end{tikzcd}$$

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 terms, hence the two reductions cannot coincide *exactly*.

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

$$\begin{tikzcd}
M_0 \arrow[rr, dashed, "*", "\iota"'] \arrow[dr, dotted] & & M_0' \arrow[r, "\beta\delta"'] \arrow[dl, dotted] & M_1 \arrow[dr, dotted] & & & M_j \arrow[rr, dashed, "*", "\iota"'] \arrow[dr, dotted] & & M_j' \arrow[r, "\beta\delta"'] \arrow[dl, dotted] & M_{j+1} \arrow[dr, dotted] & & & M_n \arrow[rr, dashed, "*", "\iota"'] \arrow[dr, dotted] & & V \arrow[r, "/" marking] \arrow[dl, dotted] & {} \\
& a_0 \arrow[rrr, "\beta\delta"'] & & & a_1 & \cdots & & a_j \arrow[rrr, "\beta\delta"'] & & & a_{j+1} & \cdots & & a_n = v \arrow[r, "/" marking] & {} &
\end{tikzcd}$$

Type erasing semantics

Forward simulation

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

Lemma (Forward simulation)

Assume $\Gamma \vdash M_1 : \tau$.
1) If $M_1 \longrightarrow_\iota M_2$, *then* $\lceil M_1 \rceil = \lceil M_2 \rceil$
2) If $M_1 \longrightarrow_{\beta\delta} M_2$, *then* $\lceil M_1 \rceil \longrightarrow_{\beta\delta} \lceil M_2 \rceil$

Type erasing semantics Backward simulation

The backward direction is more delicate to state, 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 $\Gamma \vdash M_1 : \tau$ *and* $\lceil M_1 \rceil \longrightarrow a$.
Then, there exists a term M_2 *such that* $M_1 \longrightarrow_\iota^* \longrightarrow_{\beta\delta} M_2$ *and* $\lceil M_2 \rceil = a$.

Type erasing semantics Assumption on δ-reduction

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

We need δ-reduction to be defined on type erasures.

- We may prove the theorem directly for some concrete examples of δ-reduction.
  However, keeping δ-reduction abstract is preferable to avoid repeating the same reasoning.
- We assume that it is such that type erasure establishes a bisimulation for δ-reduction taken alone.

Type erasing semantics Assumption on δ-reduction

We assume that for any explicitly-typed term M of the form $d\,\tau_1 \ldots\, \tau_j\, V_1 \ldots\, V_k$ such that $\Gamma \vdash M : \tau$, the following properties hold:

(1) If $M \longrightarrow_\delta M'$, then $\lceil M \rceil \longrightarrow_\delta \lceil M' \rceil$.

(2) If $\lceil M \rceil \longrightarrow_\delta a$, then there exists M' such that $M \longrightarrow_\delta M'$ and a is the type-erasure of M'.

Remarks

- In most cases, the assumption on δ-reduction is obvious to check.
- In general the δ-reduction on untyped terms is larger than the projection of δ-reduction on typed terms.
- If we restrict δ-reduction to implicitly-typed terms, then it usually coincides with the projection of δ-reduction of explicitly-typed terms.

Type erasing semantics Proofs

The forward simulation is straightforward.

The backward simulation can first be shown assuming that M_1 is ι-normal.

The general case follows, since then M_1 ι-reduces to a normal form M_1' preserving typings; then, the lemma can be applied to M_1' instead of M_1.

Notice that this argument relies on the termination of ι-reduction alone.

The termination of ι-reduction is easy for System F, since it strictly decreases the number of type abstractions. (In System F^ω, it requires termination of simply-typed λ-calculus.)

The proof of backward simulation uses a few helper lemmas to deal with the fact that type-erasure is not injective.

Type erasing semantics

Helper lemmas

Lemma

1) *A term that erases to* $\bar{e}[a]$ *is of the form* $\bar{E}[M]$ *where* $\lceil \bar{E} \rceil$ *is* $\bar{e}$ *and* $\lceil M \rceil$ *is* a*, and moreover,* M *does not start with a type abstraction nor a type application.*
2) *An evaluation context* $\bar{E}$ *whose erasure is the empty context is a retyping context* $\mathcal{R}$*.*
3) *If* $\mathcal{R}[M]$ *is in* ι*-normal form, then* $\mathcal{R}$ *is of the form* $\Lambda\vec{\alpha}.[]\,\vec{\tau}$*.*

Type erasing semantics

Helper lemmas

Lemma (inversion of type erasure)

Assume $\lceil M \rceil = a$

- *If* a *is* x, *then* M *is of the form* $\mathcal{R}[x]$
- *If* a *is* c, *then* M *is of the form* $\mathcal{R}[c]$
- *If* a *is* $\lambda x.\, a_1$, *then* M *is of the form* $\mathcal{R}[\lambda x\!:\!\tau.\, M_1]$ *with* $\lceil M_1 \rceil = a_1$
- *If* a *is* $a_1\ a_2$, *then* M *is of the form* $\mathcal{R}[M_1\ M_2]$ *with* $\lceil M_i \rceil = a_i$

The proof is by induction on M.

Type erasing semantics Helper lemmas

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

Assume $\Gamma \vdash M : \tau$ *and* M *is* ι*-normal. If* $\lceil M \rceil$ *is a value* v*, then* M *is a value* V*. Moreover,*

- *If* v *is* $\lambda x.\, a_1$*, then* V *is* $\Lambda \vec{\alpha}. \lambda x{:}\tau.\, M_1$ *with* $\lceil M_1 \rceil = a_1$*.*
- *If* v *is a partial application* $c\, v_1\ \ldots v_n$ *then* V *is* $\mathcal{R}[c\, \vec{\tau}\, V_1\ \ldots V_n]$ *with* $\lceil V_i \rceil = v_i$*.*

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

Corollary

Let M be a well-typed term in ι-normal form whose erasure is a.

- *If a is $(\lambda x.\, a_1)\; v$, then M if of the form $\mathcal{R}[(\lambda x{:}\tau.\, M_1)\; V]$, with $\lceil M_1 \rceil = a_1$ and $\lceil V \rceil = v$.*
- *If a is a full application $(d\; v_1 \ldots\; v_n)$, then M is of the form $\mathcal{R}[d\; \vec{\tau}\; V_1\; \ldots V_n]$ and $\lceil V_i \rceil$ is v_i.*

Type erasing semantics Backward simulation

The proof of backward simulation in the case M is ι-normal is by induction on the reduction in $\lceil F \rceil$.

Type erasing semantics Backward simulation

Case $\lceil M \rceil$ *is* $(\lambda x.\, a_1)\; v$:

By the previous corollary, M is of the form $\mathcal{R}[(\lambda x\!:\!\tau_1.\, M_1)\; V]$. Since $\mathcal{R}$ is an evaluation context, M reduces to $\mathcal{R}[[x \mapsto V]M_1]$ whose erasure is $[x \mapsto v]a_1$, *i.e.* a.

Type erasing semantics Backward simulation

Case $\lceil M \rceil$ is a full application $(d\, v_1 \ldots\, v_n)$ and reduces to a:

Then M is of the form $\mathcal{R}[M_0]$ where M_0 is $d\, \vec{\tau}\, V_1\ \ldots V_n$ and $\lceil V_i \rceil$ is v_i.

Since $\lceil M_0 \rceil \rightsquigarrow a$, by the backward assumption for δ-rules, there exists M_0' such that $M_0 \longrightarrow_\delta M_0'$ and $\lceil M_0' \rceil$ is a. Let M' be $\mathcal{R}[M_0']$. Since $\mathcal{R}$ is an evaluation context, we have $M \longrightarrow_\delta M'$ and $\lceil M' \rceil$ is a.

Type erasing semantics Backward simulation

Case $\lceil M \rceil$ *is* $e[a_1]$ *and* $a_1 \longrightarrow a_2$:

Then, M is of the form $\bar{E}[M_1]$ where $\lceil \bar{E} \rceil$ and $\lceil M_1 \rceil$ are e and a_1.

By compositionality, M_1 is well-typed.

Since M is ι-normal and $\bar{E}$ is an evaluation context, M_1 is also ι-normal.

By induction hypothesis, it $\beta\delta$-reduces to a term M_2 whose erasure is a_2.

By Rule $\textsc{Context}$, M $\beta\delta$-reduces to $\bar{E}[M_2]$ whose erasure is a.

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 a well-typed antecedent in ι-normal form in F, which, by progress in F, either $\beta\delta$-reduces or is a value; then, its type erasure $\beta\delta$-reduces (by forward simulation) or is a value (by observation).

Subject reduction Assume that $\Gamma \vdash a_1 : \tau$ and $a_1 \longrightarrow a_2$.
By well-typedness of a_1, there exists a term M_1 that erases to a_1 such that $\Gamma \vdash M_1 : \tau$.
By backward simulation in F, there exists M_2 such that and $M_1 \longrightarrow_\iota^* \longrightarrow_{\beta\delta} M_2$ and $\lceil M_2 \rceil$ is a_2.
By subject reduction in F, $\Gamma \vdash M_2 : \tau$, which implies $\Gamma \vdash a_2 : \tau$.

Type erasing semantics

The design of advanced typed systems for programming languages is usually done in explicitly-typed version, 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.

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.

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 $\mathit{ref}\,\tau_1$ and $\mathit{ref}\,\tau_2$, one could store a value of type τ_1 and read back a value of type τ_2.

Hence, it should also be unsound if a location could have type $\forall\alpha.\,\mathit{ref}\,\tau$ (where α appears in τ) as it could then be specialized to both types $\mathit{ref}\,([\alpha \mapsto \tau_1]\tau)$ and $\mathit{ref}\,([\alpha \mapsto \tau_2]\tau)$.

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

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:

$$\begin{aligned}
&\textit{let}\; y : \forall\alpha.\, \textit{ref}\,(\alpha \to \alpha) = \Lambda\alpha.\textit{ref}\,(\alpha \to \alpha)\;(\lambda z{:}\alpha.\, z)\;\textit{in}\\
&\quad (y\;\textit{bool}) := (\textit{bool} \to \textit{bool})\; not;\\
&\quad !(\textit{int} \to \textit{int})\;(y\;\textit{int})\;1 \,/\, \varnothing\\
\xrightarrow{*}\;& not\;1 \,/\, \ell \mapsto not
\end{aligned}$$

A counter example

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

$$\begin{array}{l} \textit{let}\ y : \forall\alpha.\, \textit{ref}\,(\alpha \to \alpha) = \Lambda\alpha.\textit{ref}\,(\alpha \to \alpha)\ (\lambda z{:}\alpha.\, z)\ \textit{in} \\ \quad (y\ \textit{bool}) := (\textit{bool} \to \textit{bool})\ not; \\ \quad !(\textit{int} \to \textit{int})\ (y\ \textit{int})\ 1 \,/\, \varnothing \\ \xrightarrow{*}\ not\ 1 \,/\, \ell \mapsto not \end{array}$$

What happens is that the evaluation of the reference:

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

Nailing the bug

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

$$\textsc{Context}\ \frac{\mathit{ref}\,\tau\ V \,/\, \varnothing \longrightarrow \ell \,/\, \ell \mapsto V}{\Lambda\alpha.\mathit{ref}\,\tau\ V \,/\, \varnothing \longrightarrow \Lambda\alpha.\ell \,/\, \ell \mapsto V}$$

While we have

$$\alpha \vdash \mathit{ref}\,\tau\ V \,/\, \varnothing : \mathit{ref}\,\tau \qquad \text{and} \qquad \alpha \vdash \ell \,/\, \ell \mapsto V : \mathit{ref}\,\tau$$

We have

$$\vdash \Lambda\alpha.\mathit{ref}\,\tau\ V \,/\, \varnothing : \forall\alpha.\,\mathit{ref}\,\tau \qquad \text{but not} \qquad \vdash \Lambda\alpha.\ell \,/\, \ell \mapsto V : \forall\alpha.\,\mathit{ref}\,\tau$$

Hence, the context case of subject reduction breaks.

Nailing the bug

The typing derivation of $\Lambda\alpha.\ell$ requires a store typing Σ of the form $\ell : \tau$ and a derivation of the form:

$$\textsc{Tabs}\ \frac{\Sigma, \alpha \vdash \ell : \mathit{ref}\,\tau}{\Sigma \vdash \Lambda\alpha.\ell : \forall\alpha.\,\mathit{ref}\,\tau}$$

However, the typing context Σ, α is ill-formed as α appears free in Σ.

Instead, a well-formed premise should bind α earlier as in $\alpha, \Sigma \vdash \ell : \mathit{ref}\,\tau$, but then, Rule $\textsc{Tabs}$ cannot be applied.

By contrast, the expression $\mathit{ref}\,\tau\ V$ is pure, so Σ may be empty:

$$\textsc{Tabs}\ \frac{\alpha \vdash \mathit{ref}\,\tau\ V : \mathit{ref}\,\tau}{\varnothing \vdash \mathit{ref}\,\tau\ V : \forall\alpha.\,\mathit{ref}\,\tau}$$

The expression $\Lambda\alpha.\ell$ is correctly rejected as ill-typed, so $\Lambda\alpha.(\mathit{ref}\,M\ V)$ should also be rejected. Why?

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 what happens in our example. The type variable α which appears in the type of V is abstracted in front of *ref* V.

When *ref* V reduces, $\alpha \rightarrow \alpha$ 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

$$\frac{\Gamma, \alpha \vdash u : \tau}{\Gamma \vdash \Lambda\alpha.u : \forall\alpha.\tau}$$

Value forms:

$$u ::= x \mid V \mid \Lambda\tau.u \mid u\ \tau$$

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

Subject reduction holds again.

A good intuition: internalizing configurations

A configuration $M \,/\, \mu$ is an expression M in a memory μ. The memory can be viewed as a recursive extensible record.

The configuration $M \,/\, \mu$ may be viewed as the recursive definition (of values) *let rec* $m : \Sigma = \mu$ *in* $[\ell \mapsto m.\ell]M$ where Σ 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

$$\begin{array}{rl} & \textit{let rec } m : \Sigma \qquad\quad = \mu \qquad\qquad \textit{in } E[\textit{ref}\; \tau\; V] \\ \longrightarrow & \textit{let rec } m : \Sigma, \ell : \tau = \mu, \ell \mapsto v \textit{ in } E[m.\ell] \end{array}$$

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

Otherwise, we are violating the scoping rules.

Clarifying the typing rules

Let us review the typing rules for configurations:

?

Clarifying the typing rules

Let us review the typing rules for configurations:

$$\textsc{Config}\ \frac{\vdash M : \tau \qquad \vdash \mu : \Sigma}{\vdash M \,/\, \mu : \tau} \qquad \textsc{Store}\ \frac{\forall \ell \in \mathrm{dom}(\mu), \qquad \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vdash \mu : \Sigma}$$

Clarifying the typing rules

Let us review the typing rules for configurations:

$$\textsc{Config}\ \frac{\vec{\alpha} \vdash M : \tau \qquad \vec{\alpha} \vdash \mu : \Sigma}{\vec{\alpha} \vdash M \,/\, \mu : \tau} \qquad \textsc{Store}\ \frac{\forall \ell \in \mathrm{dom}(\mu), \quad \vec{\alpha}, \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vec{\alpha} \vdash \mu : \Sigma}$$

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 $\vec{\alpha}, \Sigma, \Gamma \vdash M : \tau$ although we may see $\vec{\alpha}, \Sigma, \Gamma$ as a whole typing context Γ'.

For locations, we need a new context formation rule:

$$\textsc{WfEnvLoc}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau \qquad \ell \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \ell : \tau}$$

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

- Σ 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 $\vec{\alpha}, \Sigma, \Gamma'$)

$$\frac{\textsc{Loc}}{\Gamma \vdash \ell : \mathit{ref}\, \Gamma(\ell)}$$

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

$$\begin{array}{rcl} \mathit{ref} & : & \forall \alpha.\, \alpha \to \mathit{ref}\, \alpha \\ (!) & : & \forall \alpha.\, \mathit{ref}\, \alpha \to \alpha \\ (:=) & : & \forall \alpha.\, \mathit{ref}\, \alpha \to \alpha \to \mathit{unit} \end{array}$$

Which ones are constructors?

Clarifying the typing rules

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

$$\frac{\textsc{Loc}}{\Gamma \vdash \ell : \textit{ref}\,\Gamma(\ell)}$$

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

$$\begin{array}{rcl} \textit{ref} & : & \forall \alpha.\, \alpha \to \textit{ref}\,\alpha \\ (!) & : & \forall \alpha.\, \textit{ref}\,\alpha \to \alpha \\ (:=) & : & \forall \alpha.\, \textit{ref}\,\alpha \to \alpha \to \textit{unit} \end{array}$$

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

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

$$\begin{array}{rcll} \textit{ref}\,\tau\; V \,/\, \mu & \longrightarrow & \ell \,/\, \mu[\ell \mapsto V] & \text{if } \ell \notin \mathrm{dom}(\mu) \\ \ell := (\tau)\;\; V \,/\, \mu & \longrightarrow & () \,/\, \mu[\ell \mapsto V] & \\ !\tau\; \ell \,/\, \mu & \longrightarrow & \mu(\ell) \,/\, \mu & \end{array}$$

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 M/μ where M is a full application of constants ref, $(!)$, and $(:=)$ to types and values can always be reduced.

Theorem (Progress)

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

Consequences

The problematic program is now syntactically ill-formed:

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

Indeed, $\textit{ref}\,(\lambda z{:}\alpha.\, z)$ 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.

Therefore, this style of introducing references in System F (or in ML) is *not a conservative extension.*

Assuming:

$$map : \forall\alpha.\, \mathit{list}\,\alpha \to \mathit{list}\,\alpha \qquad id : \forall\alpha.\,\alpha \to \alpha$$

This expression is ill-typed:

$$\Lambda\alpha.map\,\alpha\,(id\,\alpha)$$

?

Consequences

With the value restriction, some pure programs become ill-typed, even though they were well-typed in the absence of references.

Therefore, this style of introducing references in System F (or in ML) is *not a conservative extension.*

Assuming:

$$map : \forall\alpha.\, \mathit{list}\, \alpha \to \mathit{list}\, \alpha \qquad id : \forall\alpha.\, \alpha \to \alpha$$

This expression is ill-typed:

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

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

$$\Lambda\alpha.\lambda y{:}\mathit{list}\,\alpha.\, map\ \alpha\ (id\ \alpha)\ y$$

?

Consequences

With the value restriction, some pure programs become ill-typed, even though they were well-typed in the absence of references.

Therefore, this style of introducing references in System F (or in ML) is *not a conservative extension.*

Assuming:

$$map : \forall\alpha.\, \mathit{list}\,\alpha \to \mathit{list}\,\alpha \qquad\qquad id : \forall\alpha.\, \alpha \to \alpha$$

This expression is ill-typed:

$$\Lambda\alpha.map\;\alpha\;(id\;\alpha)$$

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

$$\Lambda\alpha.\lambda y{:}\mathit{list}\,\alpha.\,map\;\alpha\;(id\;\alpha)\;y$$

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 will definitely 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:

- $\Lambda\alpha.((\lambda x{:}\tau.\,u)\ u)$ because the inner expression is non-expansive. $\Lambda\alpha.($*let* $x : \tau = u$ *in* $u)$, which is its syntactic sugar, as well.
- *let* $x : \forall\alpha.$ *list* $\alpha = \Lambda\alpha.(M_1\ M_2)$ *in* M because α appears only positively in the type of $M_1\ M_2$.

Objective Caml implements both refinements.

In practice

In fact, $\Lambda\alpha.M$ need only be forbidden when α appears in the type of some exposed expansive term at some negative occurrence, where exposed subterms are those that do not appear under some λ-abstraction.

For instance, the expression

$$\begin{array}{l} \textit{let}\ x : \forall\alpha.\,\textit{int} \times (\textit{list}\ \alpha) \times (\alpha \to \alpha) = \\ \quad \Lambda\alpha.(\textit{ref}\ (1+2),\ (\lambda x{:}\alpha.\,x)\ \textit{Nil},\ \lambda x{:}\alpha.\,x) \\ \ \textit{in}\ M \end{array}$$

may be accepted because α appears only in the type of the non-expansive exposed expression $\lambda x{:}\alpha.\,x$ and only positively in the type of the expansive expression $(\lambda x{:}\alpha.\,x)$ *Nil*.

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.

There is still on going research for tracing side effects more precisely, in particular to better circumvent their use.

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. This permits strong updates—updates that may also change the type of references.

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.

Pottier and Protzenko [2012] are designing a language, called Mezzo , where mutable states is tracked very precisely, using permissions, ownership, and afine types.

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 a_1 in a term of the form $[x \mapsto a_1]a_2$ using a let-binding form *let* $x = a_1$ *in* a_2 so as to avoid code duplication.

Expressions of the simply-typed λ-calculus are extended with 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 τ if and only if its *let-normal form* has type τ in the simply-typed λ-calculus.

A term's let-normal form is obtained by iterating the rewrite rule (in all context—not just evaluation contexts):

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

Notice that we use 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 τ if and only if its *let-normal form* has type τ in the simply-typed λ-calculus.

A term's let-normal form is obtained by iterating the rewrite rule (in all context—not just evaluation contexts):

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

Notice that we use 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 use the more intuitive rule:

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

Some intuitions on the definition of ML

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

?

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 forms.

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 usually presented in its implicitly-typed version.

Terms are now given by:

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

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

Note: As constants behave much as program variables from a typing point of view, and do not raise any typing issues, we omit them here for conciseness—but we still keep them in the course notes.

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:

$$\tau ::= \alpha \mid \tau \rightarrow \tau \mid \ldots$$

A separate category of *type schemes* is introduced:

$$\sigma ::= \tau \mid \forall \alpha.\, \sigma$$

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

We often write $\forall \vec{\alpha}.\, \tau$ as a short hand for $\forall \alpha_1. \ldots \forall \alpha_n.\, \tau$.

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 often introduced implicitly and not part of Γ. However, we keep our presentation where type variables are explicitly declared in Γ.

Judgments now take the form:

$$\Gamma \vdash a : \sigma$$

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

Typing rules

A standard, non-syntax-directed presentation is:

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \textsc{ml-Var} \qquad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}\ \textsc{ml-Abs} \qquad \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}\ \textsc{ml-App}$$

$$\frac{\Gamma \vdash a_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash a_2 : \sigma_2}{\Gamma \vdash \mathit{let}\ x = a_1\ \mathit{in}\ a_2 : \sigma_2}\ \textsc{ml-Let} \qquad \frac{\Gamma, \alpha \vdash a : \sigma}{\Gamma \vdash a : \forall \alpha.\, \sigma}\ \textsc{ml-Gen} \qquad \frac{\Gamma \vdash a : \forall \alpha.\, \sigma}{\Gamma \vdash a : [\alpha \mapsto \tau]\sigma}\ \textsc{eml-Inst}$$

Rules ABS and APP are as in simply-typed λ-calculus: *λ-bound variables receive a monotype.*

Rule LET moves a type scheme into the environment which VAR and INST can exploit: although syntactically unchanged, Rule VAR now returns a type scheme $\Gamma(x)$, which Rule INST may instantiate.

Rules GEN and INST are as in implicitly-typed System F. Except that *type variables are instantiated with monotypes.*

Explicitly-typed terms (*e*ML)

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

$$M ::= x \mid \lambda x{:}\tau.\, M \mid M\ M \mid \Lambda\alpha.M \mid M\ \tau \mid \textit{let}\ x : \sigma = M\ \textit{in}\ M \ldots$$

The subset *e*ML of $\lceil F \rceil$ whose type erasure is in ML is defined by:

$$\textsc{eml-Var}\quad \Gamma \vdash x : \Gamma(x)$$

$$\textsc{eml-Abs}\quad \frac{\Gamma, x : \tau_0 \vdash M : \tau}{\Gamma \vdash \lambda x{:}\tau_0.\, M : \tau_0 \to \tau}$$

$$\textsc{eml-App}\quad \frac{\Gamma \vdash M_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash M_1\ M_2 : \tau_1}$$

$$\textsc{eml-Let}\quad \frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \sigma_2}{\Gamma \vdash \textit{let}\ x : \sigma_1 = M_1\ \textit{in}\ M_2 : \sigma_2}$$

$$\textsc{eml-Tabs}\quad \frac{\Gamma, \alpha \vdash M : \sigma}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\, \sigma}$$

$$\textsc{eml-Tapp}\quad \frac{\Gamma \vdash M : \forall\alpha.\, \sigma}{\Gamma \vdash M\ \tau : [\alpha \mapsto \tau]\sigma}$$

Example

Here is a simple type derivation that exploits polymorphism:

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

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

$\textsc{Gen}$ is used above $\textsc{Let}$ (at left), and $\textsc{Inst}$ is used below $\textsc{Var}$.

In fact, we will see that every type derivation can be put in this form.

A non-example

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

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

A non-example

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

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

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 τ that must be simultaneously of the form $\mathit{int} \rightarrow \tau_1$ and $\mathit{bool} \rightarrow \tau_2$, 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 $\forall\alpha.\,\alpha \rightarrow \alpha$.

Syntax-directed presentation of ML

Explicitly-typed terms have unique derivations (and unique types).

Implicitly-typed terms have many derivations, *i.e.* many explicitly-typed terms of which they are the erasure.

Can we reduce the set of explicitly-typed terms so that implicitly-typed terms have unique derivations, *i.e.* such that their corresponding explicitly typed terms have the same skeleton?

Normalization of ML terms

We may define the subset *x*ML of *e*ML as terms in the following form:

$$\begin{array}{rcl} N \in x\mathsf{ML} & ::= & \Lambda\vec{\alpha}.Q \\ Q & ::= & x\ \vec{\tau} \mid Q\ Q \mid \lambda x\!:\!\tau.\,Q \mid \textit{let}\ x : \sigma = M\ \textit{in}\ Q \end{array}$$

and such that the arity of $\vec{\tau}$ in $x\ \vec{\tau}$ is the arity of $\vec{\alpha}$ in the type scheme $\forall\vec{\alpha}.\,\tau$ assigned to the variable x.

Syntax-directed presentation of ML

Terms of *x*ML are so constrained that their typing derivations is determined by their type-erasure. Consider the following typing-rules:

xml-TAbs
$$\frac{\Gamma, \vec{\alpha} \vdash M : M}{\Gamma \vdash \Lambda\vec{\alpha}.M : \forall\vec{\alpha}.\,\tau}$$

xml-VarInst
$$\frac{\forall\vec{\alpha}.\,\tau = \Gamma(x)}{\Gamma \vdash x\;\vec{\tau} : [\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

xml-Abs
$$\frac{\Gamma, x : \tau_0 \vdash Q : \tau}{\Gamma \vdash \lambda x{:}\tau_0.\,Q : \tau_0 \to \tau}$$

xml-App
$$\frac{\Gamma \vdash Q_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash Q_2 : \tau_2}{\Gamma \vdash Q_1\;Q_2 : \tau_1}$$

xml-LetGen
$$\frac{\Gamma, \vec{\alpha} \vdash Q_1 : \tau_1 \qquad \Gamma, x : \forall\vec{\alpha}.\,\tau_1 \vdash Q_2 : \tau_2}{\Gamma \vdash \mathit{let}\; x : \forall\vec{\alpha}.\,\tau_1 = \Lambda\vec{\alpha}.Q_1\;\mathit{in}\;Q_2 : \tau_2}$$

By construction, terms of *x*ML are a syntactic subset of *e*ML.
Moreover, if M is in *x*ML, then $\Gamma \vdash_{\mathit{x}\mathsf{ML}} M : \sigma$ implies $\Gamma \vdash_{\mathit{e}\mathsf{ML}} M : \sigma$.

Conversely, can we associate to any term of *e*ML such that $\Gamma \vdash_{\mathit{e}\mathsf{ML}} M : \sigma$ a term N of *x*ML with the same type erasure such that $\Gamma \vdash_{\mathit{x}\mathsf{ML}} M : \sigma$?

Normalization of ML terms

NORM-VAR

$$\frac{\forall\vec{\alpha}.\,\tau = \Gamma(x)}{\Gamma \vdash x : \forall\vec{\alpha}.\,\tau \Rightarrow \Lambda\vec{\alpha}.x\;\vec{\alpha}}$$

NORM-TABS

$$\frac{\Gamma, \alpha \vdash M : \sigma \Rightarrow N}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\,\sigma \Rightarrow \Lambda\alpha.N}$$

NORM-TAPP

$$\frac{\Gamma \vdash M : \forall\alpha.\,\sigma \Rightarrow \Lambda\alpha.N}{\Gamma \vdash M\;\tau : [\alpha \mapsto \tau]\sigma \Rightarrow [\alpha \mapsto \tau]N}$$

NORM-ABS

$$\frac{\Gamma, x : \tau_0 \vdash M : \tau \Rightarrow Q}{\Gamma \vdash \lambda x{:}\tau_0.\,M : \tau_0 \rightarrow \tau \Rightarrow \lambda x{:}\tau_0.\,Q}$$

NORM-APP

$$\frac{\Gamma \vdash M_1 : \tau_2 \rightarrow \tau_1 \Rightarrow Q_1 \qquad \Gamma \vdash M_2 : \tau_2 \Rightarrow Q_2}{\Gamma \vdash M_1\;M_2 : \tau_1 \Rightarrow Q_1\;Q_2}$$

NORM-LET

$$\frac{\Gamma \vdash M_1 : \sigma_1 \Rightarrow N_1 \qquad \vec{\alpha} \mathrel{\#} \Gamma, \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \forall\vec{\alpha}.\,M \Rightarrow \Lambda\vec{\alpha}.Q}{\Gamma \vdash \textit{let}\; x : \sigma_1 = M_1\;\textit{in}\; M_2 : \forall\vec{\alpha}.\,M \Rightarrow \Lambda\vec{\alpha}.\textit{let}\; x : \sigma_1 = N_1\;\textit{in}\; Q}$$

Normalization

The translation is well-defined for all *e*ML terms:

If $\Gamma \vdash_{\mathsf{eML}} M : \sigma$ holds then $\Gamma \vdash M : \sigma \Rightarrow N$.

(The proof is by induction on M and all cases are obvious.)

Moreover:

If $\Gamma \vdash M : \sigma \Rightarrow N$ holds, then $\Gamma \vdash_{\mathsf{xML}} N : \sigma$ and $\lceil M \rceil = \lceil N \rceil$.

The proof is also by induction on M. The preservation of erasure is immediate.

The only non obvious cases for well-typedness of N are Norm-Tapp, which performs strong ι-reduction and uses type substitution, and Norm-Let, which extrudes type abstractions.

Syntax-directed presentation for ML

By dropping type information in terms from *x*ML, we thus obtain an equivalent syntax-directed presentation of ML typing rules:

$$\textsc{sml-VarInst}\quad \frac{\forall\vec{\alpha}.\,\tau = \Gamma(x)}{\Gamma \vdash_s x : [\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

$$\textsc{sml-LetGen}\quad \frac{\Gamma, \vec{\alpha} \vdash_s a_1 : \tau_1 \qquad \Gamma, x : \forall\vec{\alpha}.\,\tau_1 \vdash_s a_2 : \tau_2}{\Gamma \vdash_s \textit{let}\ x = a_1\ \textit{in}\ a_2 : \tau_2}$$

$$\textsc{sml-Abs}\quad \frac{\Gamma, x : \tau_0 \vdash_s a : \tau}{\Gamma \vdash_s \lambda x.\,a : \tau_0 \to \tau}$$

$$\textsc{sml-App}\quad \frac{\Gamma \vdash_s a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash_s a_2 : \tau_2}{\Gamma \vdash_s a_1\ a_2 : \tau_1}$$

$$\textsc{sml-TAbs}\quad \frac{\Gamma, \vec{\alpha} \vdash_s M : M}{\Gamma \vdash_s \Lambda\vec{\alpha}.M : \forall\vec{\alpha}.\,\tau}$$

Then, the judgments $\Gamma \vdash a : \sigma$ and $\Gamma \vdash_{s\mathsf{ML}} a : \sigma$ 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:

$$\textsc{iml-VarInst}\quad \frac{\forall\vec{\alpha}.\,\tau = \Gamma(x)}{\Gamma \vdash x : [\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

$$\textsc{iml-LetGen}\quad \frac{\Gamma \vdash a_1 : \tau_1 \qquad \vec{\alpha} \mathrel{\#} \Gamma \qquad \Gamma, x : \forall\vec{\alpha}.\,\tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash \mathit{let}\ x = a_1\ \mathit{in}\ a_2 : \tau_2}$$

$$\textsc{iml-Abs}\quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}$$

$$\textsc{iml-App}\quad \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}$$

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

Lemma (Substitution lemma for types)

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

Type soundness for ML

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

This also implies that progress holds in ML.

However, this does not imply that subject reduction holds for ML. Indeed, ML expressions could reduce to System F expressions that are not in the ML subset.

Most proofs of subject reduction for ML use implicitly-typed terms. For instance, see Wright and Felleisen [1994], Pottier and Rémy [2005].

Subject reduction in *e*ML

The proof of subject reduction follows the same schema as for System F.

The main part of the proof also works almost unchanged.

However, it uses auxiliary lemmas (inversion, permutation, weakening, type substitution, term substitution, compositionality) that all need to be rechecked, since those lemmas conclude with typing judgments in F that may not necessarily hold in *e*ML.

Unsurprisingly, all proofs can be easily adjusted.

See also the course notes for an indirect proof reusing subject reduction in System F.

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 Jonathan Protzenko. Programming with permissions in Mezzo. Submitted for publication, October 2012.

- ▷ 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.

Bibliography V

- ▷ 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.
- ▷ 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.

Bibliography VI

- ▷ 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.
- ▷ 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.

Bibliography VII

- 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.