MPRI 2.4, Functional programming and type systems
Metatheory of System F

Didier Rémy

Bienvenue, Welcome!

English or French?

Questions must be asked in the language your speak best (French by default)

Online material: regularly visit the course page!

https://gitlab.inria.fr/fpottier/mpri-2.4-public/blob/master/README.md

The course is composed of 4 parts, 5 lessons each, not splitable

1. Metatheory of typed programming languages
2. Interpretation, compilation, and program transformations
3. Typed-directed programming
4. Rust: programming safely with resources

Questions are welcome!

- Anytime! During the lesson, at the breaks, by email: Didier.Remy@inria.fr
- But, don't wait until the end of the course ! That will be too late

You are there to learn and **. . .** **we are here to help You!**

Some of you may find the course difficult. . .

- do the exercises, check the corrections, ask us if you can't do them.
- discuss with us, and. . . the earlier the better!
- don't wait until the exams! . . . **but you can all pass**

Evaluation $\approx$ **Partial exam + Final exam + Programming task / 3**

- Given after the partial exam, due by the end of January *Mandatory*

Questions?

Plan of the course

Metatheory of System F

ADTs, Recursive types, Existential types, GATDs

Going higher order with F^ω!

Logical relations

Side effects, References, Value restriction

Type reconstruction

Overloading

Metatheory of System F

Proofs

Since 2017-2018, this course is shorter: you can see extra material in courses notes (and in slides of year 2016).

Detailed proofs of main results are not shown in class anymore, but are still part of the course:

You are supposed to read, understand them.
and be able to reproduce them.

Formalization of System F is a basic. You *must* master it.

Some of the metatheory will be done in Coq, by François, Pottier,

—for your help or curiosity,

What are types?

- Types are:
  "a concise, formal description of the behavior of a program fragment."
- Types must be *sound*:
  programs must behave as prescribed by their types.
- Hence, types must be *checked* and ill-typed programs must be rejected.

What are they useful for?

- Types serve as *machine-checked* documentation.
- Data types help *structure* programs.
- Types provide a *safety* guarantee.
- Types can be used to drive *compiler optimizations.*
- Types encourage *separate compilation*, *modularity*, and *abstraction*.

Type-preserving compilation

Types make sense in *low-level* programming languages as well—even *assembly languages* can be statically typed! [Morrisett et al., 1999]

In a *type-preserving* compiler, every intermediate language is typed, and every compilation phase maps typed programs to typed programs.

Preserving types provides insight into a transformation, helps *debug* it, and paves the way to a *semantics preservation* proof [Chlipala, 2007].

Interestingly enough, lower-level programming languages often require richer type systems than their high-level counterparts.

Typed or untyped?

Reynolds [1985] nicely sums up a long and rather acrimonious debate:

> *"One side claims that untyped languages preclude compile-time error checking and are succinct to the point of unintelligibility, while the other side claims that typed languages preclude a variety of powerful programming techniques and are verbose to the point of unintelligibility."*

The issues are *safety*, *expressiveness*, and *type inference*.

Typed, Sir! with better types.

In fact, Reynolds settles the debate:

> *"From the theorist's point of view, both sides are right, and their arguments are the motivation for seeking type systems that are more flexible and succinct than those of existing typed languages."*

Today, the question is more whether

- to stay with rather *simple polymorphic types* (ML, System F, or F^ω).
- use more *sophisticated types* (dependent types, afine types, capabililties and ownership, effects, logical assertions, *etc.*), or
- even towards full *program proofs*!

The community is still between *programming with dependent types to capture fine invariants*, or programming with simpler types and developing *program proofs on the side* that these invariants hold —with often a preference for the latter.

Contents

Why λ-calculus?

In this course, the underlying programming language is the λ-calculus.

The λ-calculus supports *natural* encodings of many programming languages [Landin, 1965], and as such provides a suitable setting for studying type systems.

Following Church's thesis, any Turing-complete language can be used to encode any programming language. However, these encodings might not be natural or simple enough to help us in understanding their typing discipline.

Using λ-calculus, most of our results can also be applied to other languages (Java, assembly language, *etc.*).

Simply typed λ-calculus

Why?

- used to introduce the main ideas, in a simple setting
- we will then move to System F
- *still used in some theoretical studies*
- *is the language of kinds for* F^ω

Types are:

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

Terms are:

$$M ::= x \mid \lambda x\!:\!\tau.\, M \mid M\ M \mid \ldots$$

The dots are place holders for future extensions of the language.

Binders, α-conversion, and substitutions

$\lambda x{:}\tau.\, M$ *binds* variable x in M.

We write $\mathrm{fv}(M)$ for the set of free (term) variables of M:

$$\begin{aligned} \mathrm{fv}(x) &\triangleq \{x\} \\ \mathrm{fv}(\lambda x{:}\tau.\, M) &\triangleq \mathrm{fv}(M) \setminus \{x\} \\ \mathrm{fv}(M_1\ M_2) &\triangleq \mathrm{fv}(M_1) \cup \mathrm{fv}(M_2) \end{aligned}$$

We write $x \# M$ for $x \notin \mathrm{fv}(M)$.

Terms are considered equal up to renaming of bound variables:

- $\lambda x_1{:}\tau_1.\, \lambda x_2{:}\tau_2.\, x_1\ x_2$ and $\lambda y{:}\tau_1.\, \lambda x{:}\tau_2.\, y\ x$ are really the same term!
- $\lambda x{:}\tau.\, \lambda x{:}\tau.\, M$ is equal to $\lambda y{:}\tau.\, \lambda x{:}\tau.\, M$ when $y \notin \mathrm{fv}(M)$.

Substitution:

$[x \mapsto N]M$ is the capture avoiding substitution of N for x in M.

Dynamic semantics

We use a *small-step operational* semantics.

We choose a *call-by-value* variant. When adding *references*, exceptions, or other forms of side effects, this choice matters.

Otherwise, most of the type-theoretic machinery applies to call-by-name or call-by-need just as well.

Weak *v.s.* full reduction (parenthesis)

Calculi are often presented with a full reduction semantics, *i.e.* where reduction may occur in *any* context. The reduction is then non-deterministic (there are many possible reduction paths) but the calculus remains deterministic, since reduction is confluent.

Programming languages use weak reduction strategies, *i.e.* reduction is never performed under λ-abstractions, for efficiency of reduction, to have a deterministic semantics in the presence of side effects—and a well-defined cost model.

Still, type systems are usually also sound for full reduction strategies (with some care in the presence of side effects or empty types).

Type soundness for full reduction is a stronger result.

It implies that potential errors may not be hidden under λ-abstractions (this is usually true—it is true for λ-calculus and System F—but not implied by type soundness for a weak reduction strategy.)

Dynamic semantics

In the pure, explicitly-typed call-by-value λ-calculus, the *values* are the functions:

$$V ::= \lambda x\!:\!\tau.\, M \mid \ldots$$

The *reduction relation* $M_1 \longrightarrow M_2$ is inductively defined:

$$\beta_v \quad (\lambda x\!:\!\tau.\, M)\; V \longrightarrow [x \mapsto V]M \qquad\qquad \text{CONTEXT}\ \frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}$$

Evaluation contexts are defined as follows:

$$E ::= [\,]\; M \mid V\; [\,] \mid \ldots$$

We only need evaluation contexts of depth one, using repeated applications of Rule CONTEXT.

An evaluation context of arbitrary depth can be defined as:

$$\bar{E} ::= [\,] \mid E[\bar{E}]$$

Static semantics

Technically, the type system is a 3-place predicate, whose instances are called *typing judgments*, written:

$$\Gamma \vdash M : \tau$$

where Γ is a typing context.

Typing context, notations

A *typing context* (also called a *type environment*) Γ binds program variables to types.

We write $\varnothing$ for the empty context and $\Gamma, x : \tau$ for the extension of Γ with $x \mapsto \tau$.

To avoid confusion, we require $x \notin \mathrm{dom}(\Gamma)$ when we write $\Gamma, x : \tau$.

Bound variables in source programs can always be suitably renamed to avoid name clashes.

A typing context can then be thought of as a finite function from program variables to their types.

We write $\mathrm{dom}(\Gamma)$ for the set of variables bound by Γ and $x : \tau \in \Gamma$ to mean $x \in \mathrm{dom}(\Gamma)$ and $\Gamma(x) = \tau$.

Static semantics

Typing judgments are defined inductively by the following set of *inferences rules*:

$$\textsc{Var}\qquad \Gamma \vdash x : \Gamma(x)$$

$$\textsc{Abs}\qquad \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1 . M : \tau_1 \to \tau_2}$$

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

Notice that the specification is extremely simple.

In the simply-typed λ-calculus, the definition is *syntax-directed*. This is not true of all type systems.

Example

The following is a valid *typing derivation*:

$$
\dfrac{
  \dfrac{
    \dfrac{}{\Gamma \vdash f : \tau \to \tau'}\textsc{Var}
    \quad
    \dfrac{}{\Gamma \vdash x_1 : \tau}\textsc{Var}
  }{\Gamma \vdash f\ x_1 : \tau'}\textsc{App}
  \qquad
  \dfrac{
    \dfrac{}{\Gamma \vdash f : \tau \to \tau'}\textsc{Var}
    \quad
    \dfrac{}{\Gamma \vdash x_2 : \tau}\textsc{Var}
  }{\Gamma \vdash f\ x_2 : \tau'}\textsc{App}
}{
  \dfrac{f : \tau \to \tau', x_1 : \tau, x_2 : \tau \vdash (f\ x_1, f\ x_2) : \tau' \times \tau'}{\varnothing \vdash \lambda f : \tau \to \tau'.\, \lambda x_1 : \tau.\, \lambda x_2 : \tau.\, (f\ x_1, f\ x_2) : (\tau \to \tau') \to \tau \to \tau \to (\tau' \times \tau')}\textsc{Abs}
}\textsc{Pair}
$$

Γ stands for $(f : \tau \to \tau', x_1 : \tau, x_2 : \tau)$. Rule Pair is introduced later on.

Observe that:

- this is in fact, the only typing derivation (in the empty environment).
- this derivation is valid for any choice of τ and τ'
  (which in our setting are part of the source term)

Conversely, every derivation for this term must have this shape, actually be exactly this one, up to the name of variables.

Inversion of typing rules

The inversion Lemma states formally the previous informal reasoning.
It describes how the subterms of a well-typed term can be typed.

Lemma (Inversion of typing rules)

Assume $\Gamma \vdash M : \tau$.

- *If* M *is a variable* x, *then* $x \in \mathrm{dom}(\Gamma)$ *and* $\Gamma(x) = \tau$.
- *If* M *is* $M_1\ M_2$ *then* $\Gamma \vdash M_1 : \tau_2 \to \tau$ *and* $\Gamma \vdash M_2 : \tau_2$ *for some type* τ_2.
- *If* M *is* $\lambda x{:}\tau_2.\, M_1$, *then* τ *is of the form* $\tau_2 \to \tau_1$ *and* $\Gamma, x : \tau_2 \vdash M_1 : \tau_1$.

The inversion lemma is a basic property that is used in many places when reasoning by induction on terms. Although trivial in our simple setting, stating it explicitly avoids informal reasoning in proofs.

In more general settings, this may be a difficult lemma that requires reorganizing typing derivations.

Uniqueness of typing derivations

Since typing rules are syntax-directed, the shape of the derivation tree is fully determined by the shape of the term.

In our simple setting, each term has actually a unique type.
Hence, typing derivations are unique, up to the typing context.
The proof, by induction on the structure of terms, is straightforward.

Explicitly-typed terms can thus be used to describe and manipulate typing derivations (up to the typing context) in a precise and concise way.

This enables reasoning by induction on terms instead of on typing derivations, which is often lighter.

Lacking this convenience, typing derivations must otherwise be described in the meta-language of mathematics.

Explicitly *v.s.* implicitly typed?

Our presentation of simply-typed λ-calculus is *explicitly typed* (we also say in *church-style*), as parameters of abstractions are annotated with their types.

Simply-typed λ-calculus can also be *implicitly typed* (we also say in *curry-style*) when parameters of abstractions are left unannotated, as in the pure λ-calculus.

Of course, the existence of syntax-directed typing rules depends on the amount of type information present in source terms and can be easily lost if some type information is left implicit.

In particular, typing rules for terms in curry-style are not syntax-directed.

Type erasure

We may translate explicitly-typed expressions into implicitly-typed ones by dropping type annotations. This is called *type erasure*.

We write $\lceil M \rceil$ for the type erasure of M, which is defined by structural induction on M:

$$\begin{array}{rcl} \lceil x \rceil & \triangleq & x \\ \lceil \lambda x : \tau . M \rceil & \triangleq & \lambda x . \lceil M \rceil \\ \lceil M_1 \; M_2 \rceil & \triangleq & \lceil M_1 \rceil \, \lceil M_2 \rceil \end{array}$$

Type reconstruction

Conversely, can we convert implicitly-typed expressions back into explicitly-typed ones, that is, can we reconstruct the missing type information?

This is equivalent to finding a typing derivation for implicitly-typed terms. It is called *type reconstruction* (or *type inference*).
(See the course on type reconstruction.)

Type reconstruction . . . may be partial

Annotating programs with types can lead to redundancy.

Types can even become extremely cumbersome when they have to be explicitly and repeatedly provided. In some pathological cases, *type information may grow in square of the size* of the underlying untyped expression.

This creates a need for a certain degree of *type reconstruction* (also called type inference), even when the language is meant to be explicitly typed, where the source program may contain some but not all type information.

Full type reconstruction is undecidable for expressive type systems.

Some type annotations are required or type reconstruction is incomplete.

Untyped semantics

Observe that although the reduction carries types at runtime, types do not actually contribute to the reduction.

Intuitively, the semantics of terms is the same as that of their type erasures. We say that the semantics is *untyped* or *type-erasing*.

But how can we say that the semantics of typed and untyped terms coincide when these terms do not live in the same world?

By showing that the reductions in the two languages can be put into close correspondence.

Untyped semantics

Obsviously, type erasure preserves reduction.

$$\begin{tikzcd} M_1 \arrow[r, "\beta"] \arrow[d, dashed, "\lceil\rceil"'] & M_2 \arrow[d, dashed, "\lceil\rceil"] \\ a_1 \arrow[r, dashed, "\beta"'] & a_2 \end{tikzcd}$$

Lemma (Direct simulation)

If $M_1 \longrightarrow M_2$ *then* $\lceil M_1 \rceil \longrightarrow \lceil M_2 \rceil$.

Conversely, a reduction step after type erasure could also have been performed on the term before type erasure.

$$\begin{tikzcd} M_1 \arrow[r, dashed, "\beta"] \arrow[d, "\lceil\rceil"'] & M_2 \arrow[d, dashed, "\lceil\rceil"] \\ a_1 \arrow[r, "\beta"'] & a_2 \end{tikzcd}$$

Lemma (Inverse simulation)

If $\lceil M \rceil \longrightarrow a$ *then there exists* M' *such that* $M \longrightarrow M'$ *and* $\lceil M' \rceil = a$.

What we have established is a *bisimulation* between explicitly-typed terms and implicitly-typed ones.

In general, there may be reduction steps on source terms that involved only types and have no counter-part (and disappear) on compiled terms.

Untyped semantics

It is an important property for a language to have an untyped semantics.

It then has an implicitly-typed presentation.

The metatheoretical study is often easier with explicitly-typed terms, in particular when proving syntactic properties.

Properties of the implicitly-typed presentation can often be indirectly proved via an explicitly-typed presentation of the language.

This is the path we choose in this course.

(Once we have shown that implicit and explicit presentations coincide, we can choose whichever view is more convenient.)

Contents

Stating type soundness

What is a formal statement of the slogan

"Well-typed expressions do not go wrong"

By definition, a closed term M is *well-typed* if it admits some type τ in the empty environment.

By definition, a closed, irreducible term is either a value or *stuck*. Thus, a closed term can only:

- *diverge*,
- *converge* to a value, or
- *go wrong* by reducing to a stuck term.

Type soundness: the last case is not possible for well-typed terms.

Stating type soundness

The slogan now has a formal meaning:

Theorem (Type soundness)

Well-typed expressions do not go wrong.

Proof.

By Subject Reduction and Progress. □

Note *We only give the proof schema here, as the same proof will be carried again with more details in the (more complex) case of System F. —See the course notes for detailed proofs.*

Establishing type soundness

We use the syntactic proof method of Wright and Felleisen [1994].
Type soundness follows from two properties:

Theorem (Subject reduction)

Reduction preserves types: if $M_1 \longrightarrow M_2$ *then for any type* τ *such that* $\varnothing \vdash M_1 : \tau$, *we also have* $\varnothing \vdash M_2 : \tau$.

Theorem (Progress)

A (closed) well-typed term is either a value or reducible:
if $\varnothing \vdash M : \tau$ *then there exists* M' *such that* $M \longrightarrow M'$, *or* M *is a value.*

Equivalently, we may say: *closed, well-typed, irreducible terms are values.*

Contents

Adding a unit

The simply-typed λ-calculus is modified as follows. Values and expressions are extended with a nullary constructor () (read "unit"):

$$M ::= \ldots \mid () \qquad\qquad V ::= \ldots \mid ()$$

No new reduction rule is introduced.

Types are extended with a new constant *unit* and a new typing rule:

$$\tau ::= \ldots \mid \mathit{unit} \qquad\qquad \frac{\textsc{Unit}}{\Gamma \vdash () : \mathit{unit}}$$

Pairs

The simply-typed λ-calculus is modified as follows.

Values, expressions, evaluation contexts are extended:

$$\begin{array}{rcl} M & ::= & \ldots \mid (M, M) \mid \textit{proj}_i\ M \\ E & ::= & \ldots \mid ([], M) \mid (V, []) \mid \textit{proj}_i\ [] \\ V & ::= & \ldots \mid (V, V) \\ i & \in & \{1, 2\} \end{array}$$

A new reduction rule is introduced:

$$\textit{proj}_i\ (V_1, V_2) \longrightarrow V_i$$

Pairs

Types are extended:

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

Two new typing rules are introduced:

$$\textsc{Pair}\ \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2} \qquad\qquad \textsc{Proj}\ \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathit{proj}_i\ M : \tau_i}$$

Sums

Values, expressions, evaluation contexts are extended:

$$\begin{array}{rcl} M & ::= & \ldots \mid \mathit{inj}_i\ M \mid \mathit{case}\ M\ \mathit{of}\ V \mathbin{[\!]} V \\ E & ::= & \ldots \mid \mathit{inj}_i\ [] \mid \mathit{case}\ []\ \mathit{of}\ V \mathbin{[\!]} V \\ V & ::= & \ldots \mid \mathit{inj}_i\ V \\ i & \in & \{1, 2\} \end{array}$$

A new reduction rule is introduced:

$$\mathit{case}\ \mathit{inj}_i\ V\ \mathit{of}\ V_1 \mathbin{[\!]} V_2 \longrightarrow V_i\ V$$

Sums

Types are extended:

$$\tau ::= \ldots \mid \tau + \tau$$

Two new typing rules are introduced:

$$\textsc{Inj}\quad \frac{\Gamma \vdash M : \tau_i}{\Gamma \vdash \textit{inj}_i\ M : \tau_1 + \tau_2}$$

$$\textsc{Case}\quad \frac{\Gamma \vdash M : \tau_1 + \tau_2 \qquad \Gamma \vdash V_1 : \tau_1 \rightarrow \tau \qquad \Gamma \vdash V_2 : \tau_2 \rightarrow \tau}{\Gamma \vdash \textit{case}\ M\ \textit{of}\ V_1 \mathbin{[\!]} V_2 : \tau}$$

Sums with unique types

Notice that a property of simply-typed λ-calculus is lost: expressions do not have unique types anymore, *i.e.* the type of an expression is no longer determined by the expression.

Uniqueness of types can be recovered by using a type annotation in injections:

$$V ::= \ldots \mid \mathit{inj}_i\, V \text{ as } \tau$$

and modifying the typing rules and reduction rules accordingly.

Exercise

Describe an extension with the option type.

Modularity of extensions

The three preceding extensions are very similar. Each one introduces:

- a new type constructor, to classify values of a new shape;
- new expressions, to *construct* and *destruct* values of a new shape.
- new typing rules for new forms of expressions;
- new reduction rules, to specify how values of the new shape can be destructed;
- new evaluation contexts—but just to propagate reduction under the new constructors.

Subject reduction is preserved because types are preserved by the new reduction rules.

Progress is preserved because the type system ensures that the new destructors can only be applied to values such that at least one of the new reduction rules applies.

Modularity of extensions

These extensions are independent: they can be added to the λ-calculus alone or mixed altogether.

Indeed, no assumption about other extensions (the "...") is ever made, except for the classification lemma which requires, informally, that values of other shapes have types of other shapes.

This is indeed the case in the extensions we have presented: the unit has the Unit type, pairs have product types, sums have sum types.

In fact, these extensions could have been presented as several instances of a more general extension of the λ-calculus with constants, for which type soundness can be established uniformly under reasonable assumptions relating the given typing rules and reduction rules for constants.

See the treatment of *data types* in System F in the following section.

Recursive functions

The simply-typed λ-calculus is modified as follows.

Values and expressions are extended:

$$\begin{array}{rcl} M & ::= & \ldots \mid \mu f\!:\!\tau.\,\lambda x.M \\ V & ::= & \ldots \mid \mu f\!:\!\tau.\,\lambda x.M \end{array}$$

A new reduction rule is introduced:

$$(\mu f\!:\!\tau.\,\lambda x.M)\; V \longrightarrow [f \mapsto \mu f\!:\!\tau.\,\lambda x.M][x \mapsto V]M$$

Recursive functions

Types are *not* extended. We already have function types.

What does this imply as a corollary?

— Types will not distinguish functions from recursive functions.

A new typing rule is introduced:

$$\frac{\textsc{FixAbs} \qquad \Gamma, f : \tau_1 \to \tau_2 \vdash \lambda x{:}\tau_1.\, M : \tau_1 \to \tau_2}{\Gamma \vdash \mu f{:}\tau_1 \to \tau_2.\, \lambda x.M : \tau_1 \to \tau_2}$$

In the premise, the type $\tau_1 \to \tau_2$ serves both as an assumption and a goal. This is a typical feature of recursive definitions.

A derived construct: let

The construct "*let* $x : \tau = M_1$ *in* M_2" can be viewed as syntactic sugar for the β-redex "$(\lambda x{:}\tau.\, M_2)\ M_1$".

The latter can be type-checked *only* by a derivation of the form:

$$\dfrac{\textsc{Abs}\ \dfrac{\Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, M_2 : \tau_1 \to \tau_2} \qquad \Gamma \vdash M_1 : \tau_1}{\Gamma \vdash (\lambda x{:}\tau_1.\, M_2)\ M_1 : \tau_2}\ \textsc{App}$$

This means that the following *derived rule* is sound and *complete*:

$$\textsc{LetMono}\quad \dfrac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \textit{let}\ x : \tau_1 = M_1\ \textit{in}\ M_2 : \tau_2}$$

The construct "$M_1; M_2$" can in turn be viewed as syntactic sugar for *let* $x : \textit{unit} = M_1$ *in* M_2 where $x \notin \mathrm{ftv}(M_2)$.

A derived construct: let or a primitive one?

In the derived form *let* $x : \tau_1 = M_1$ *in* M_2 the type of M_1 must be explicitly given, although by uniqueness of types, it is entirely determined by the expression M_1 itself. Hence, it seems redundant.

Indeed, we can replace the derived form by a primitive form *let* $x = M_1$ *in* M_2 with the following primitive typing rule.

$$\frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \textit{let } x = M_1 \textit{ in } M_2 : \tau_2}\ \textsc{LetMono}$$

This seems better—not necessarily, because removing redundant type annotations is the task of type reconstruction and we should not bother (too much) about it in the explicitly-typed version of the language.

Minimizing the number of language constructs is at least as important as avoiding extra type annotations *in an explicitly-typed* language.

A derived construct: let rec

The construct "*let rec* $(f : \tau)\ x = M_1$ *in* M_2" can be viewed as syntactic sugar for "*let* $f = \mu f{:}\tau.\,\lambda x.M_1$ *in* M_2". The latter can be type-checked *only* by a derivation of the form:

$$\dfrac{\textsc{FixAbs}\ \dfrac{\Gamma, f : \tau \to \tau_1; x : \tau \vdash M_1 : \tau_1}{\Gamma \vdash \mu f{:}\tau \to \tau_1.\,\lambda x.M_1 : \tau \to \tau_1} \qquad \Gamma, f : \tau \to \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \textit{let } f = \mu f{:}\tau \to \tau_2.\,\lambda x.M_1 \textit{ in } M_2 : \tau_2}\ \textsc{LetMono}$$

This means that the following *derived rule* is sound and *complete*:

$$\textsc{LetRecMono}\quad \dfrac{\Gamma, f : \tau \to \tau_1; x : \tau \vdash M_1 : \tau_1 \qquad \Gamma, f : \tau \to \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \textit{let rec } (f : \tau \to \tau_1)\ x = M_1 \textit{ in } M_2 : \tau_2}$$

Contents

What is polymorphism?

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

Why polymorphism?

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

In short, it should have polymorphic type:

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

which *instantiates* to the monomorphic types:

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

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? 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 *one* inhabitant, up to $\beta\eta$-equivalence, namely the identity.

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 (including, 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.

For instance, the polymorphic type $\forall\alpha.\,\alpha \to \alpha$ has only *one* inhabitant, up to $\beta\eta$-equivalence, namely the identity.

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 (including, e.g., a function that sorts in reverse order, or a function that removes duplicates)

Ad hoc v.s. parametric polymorphism

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 *v.s.* parametric polymorphism: first definition

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

$$\begin{gathered} \mathit{int} \rightarrow \mathit{int}, \\ \mathit{bool} \rightarrow \mathit{bool}, \\ \dots \\ \forall \alpha. \alpha \rightarrow \alpha \end{gathered}$$

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

$$\begin{gathered} \mathit{int} \rightarrow \mathit{int} \rightarrow \mathit{int}, \\ \mathit{string} \rightarrow \mathit{string} \rightarrow \mathit{string}, \\ \dots \\ \textit{but not} \\ \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \end{gathered}$$

Ad hoc *v.s.* parametric polymorphism: 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 *v.s.* 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 the case of Haskell type classes, the two views can be reconciled. (See the course on overloading.)

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

Contents

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* (syntactic) 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:

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

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

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

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

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

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

Note

Rule $\textsc{WfEnvVar}$ need not the premise $\vdash \Gamma$, which follows from $\Gamma \vdash \tau$

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 will remind implicit well-formedness premises in the definition of typing rules.)*

Type-passing semantics

We need the following reduction for type-level expressions:

$$(\Lambda\alpha.\, M)\ \tau \longrightarrow [\alpha \mapsto \tau]M \tag{ι}$$

Then, there is a choice.

Historically, in most presentations of System F, type abstraction stops the evaluation. It is described by:

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

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:

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

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:

  if f is list flattening of type $\forall\alpha.\,\mathit{list}\,(\mathit{list}\,\alpha) \rightarrow \mathit{list}\,\alpha$, 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.
- *for this very reason*, 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 is 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 ML subset.

Then, both presentations can be used, interchangeably.

System F, full definition (on one slide) To remember!

Syntax

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall\alpha.\tau$$
$$M ::= x \mid \lambda x{:}\tau.\, M \mid M\ M \mid \Lambda\alpha.\, M \mid M\ \tau$$

Typing rules

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

Abs
$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, M : \tau_1 \to \tau_2}$$

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

App
$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\ M_2 : \tau_2}$$

Tapp
$$\frac{\Gamma \vdash M : \forall\alpha.\tau}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}$$

Semantics

$$V ::= \lambda x{:}\tau.\, M \mid \Lambda\alpha.\, V$$
$$E ::= [\,]\ M \mid V\ [\,] \mid [\,]\ \tau \mid \Lambda\alpha.\, [\,]$$

$$(\lambda x{:}\tau.\, M)\ V \longrightarrow [x \mapsto V]M$$
$$(\Lambda\alpha.\, V)\ \tau \longrightarrow [\alpha \mapsto \tau]V$$

Context
$$\frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}$$

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:

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

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

Sum and inductive types such as Natural numbers, List, *etc.* can also be encoded.

Primitive data-structures as 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 in 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 then generalize it to arbitrary constructors and destructors.)

Constructors and destructors Pairs

Types are extended with a type constructor $\times$ 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?

- 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 for subject reduction but replaced by a stronger proof obligation for 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 $G \in \mathcal{G}$, with their arity *arity* (G). We assume that types respect the arities of type constructors.

Given G, a type of the form $G\,(\vec{\tau})$ is called a G-type.
A type τ is called a *datatype* if it is a G-type for some type constructor G.

For instance $\mathcal{G}$ is $\{\mathit{unit}, \mathit{int}, \mathit{bool}, (\_ \times \_), \mathit{list}\ \_, \ldots\}$

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

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

We require that

- $\boxed{\tau}$ be a datatype whenever c is a constructor (key for progress);
- the arity of destructors be strictly positive
  (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:

$$\begin{array}{rcl} M & ::= & \ldots \mid c \\ c & ::= & C \mid d \end{array} \qquad \qquad \frac{\textsc{Cst} \quad c : \tau \in \Delta}{\Gamma \vdash c : \tau}$$

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

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

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 — Soundness requirements

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:

- Well-typed full applications of destructors can be reduced

  If $\vec{\alpha} \vdash M_1 : \tau$ and M_1 is of the form $d\ \tau_1\ \ldots\ \tau_k\ V_1\ \ldots\ V_{\textsf{arity}(d)}$ then there exists M_2 such that $M_1 \longrightarrow M_2$.

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

Example

Unit

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 also perfectly fits 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:

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

(by inspection of the typing derivation)

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.

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:

$$\begin{array}{l} \mathit{matchlist}\,\tau_1\,\tau_2\,(\mathit{Nil}\,\tau)\,V_n\,V_c \longrightarrow V_n \\ \mathit{matchlist}\,\tau_1\,\tau_2\,(\mathit{Cons}\,\tau\,V_h\,V_t)\,V_n\,V_c \longrightarrow V_c\,V_h\,V_t \end{array}$$

2) See the case of pairs in the course.

Contents

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 (new)
- compositionality

Inversion of typing judgements

Lemma (Inversion of typing rules)

Assume $\Gamma \vdash M : \tau$.

- *If* M *is a variable* x, *then* $x \in \mathrm{dom}(\Gamma)$ *and* $\Gamma(x) = \tau$.
- *If* M *is* $\lambda x{:}\tau_0.\, M_1$, *then* τ *is of the form* $\tau_0 \to \tau_1$ *and* $\Gamma, x : \tau_0 \vdash M_1 : \tau_1$.
- *If* M *is* $M_1\ M_2$, *then* $\Gamma \vdash M_1 : \tau_2 \to \tau$ *and* $\Gamma \vdash M_2 : \tau_2$ *for some type* τ_2.
- *If* M *is a constant* c, *then* $c \in \mathrm{dom}(\Delta)$ *and* $\Delta(x) = \tau$.
- *If* M *is* $M_1\ \tau_2$ *then* τ *is of the form* $[\alpha \mapsto \tau_2]\tau_1$ *and* $\Gamma \vdash M_1 : \forall\alpha.\, \tau_1$.
- *If* M *is* $\Lambda\alpha.\, M_1$, *then* τ *is of the form* $\forall\alpha.\, \tau_1$ *and* $\Gamma, \alpha \vdash M_1 : \tau_1$.

The inversion lemma is a basic property that is used in many places when reasoning by induction on terms. It may not always be as trivial as in our simple setting: stating it explicitly avoids informal reasoning in proofs.

Type soundness — Weakening

Lemma (Weakening)

Assume $\Gamma \vdash M : \tau$.

1) If $x \# \Gamma$ *and* $\Gamma \vdash \tau'$, *then* $\Gamma, x : \tau' \vdash M : \tau$

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

That is, if $\vdash \Gamma, \Gamma'$, *then* $\Gamma, \Gamma' \vdash M : \tau$.

The proof is by induction on M, then by cases on M applying the inversion lemma.

Cases for value and type abstraction appeal to the permutation lemma:

Lemma (Permutation)

If $\Gamma, \Gamma_1, \Gamma_2, \Gamma' \vdash M : \tau$ *and* $\Gamma_1 \# \Gamma_2$ *then* $\Gamma, \Gamma_2, \Gamma_1, \Gamma' \vdash M : \tau$.

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 by induction on M.

The case for type and value abstraction requires the strengthened version with an arbitrary context Γ'. The proof is then straightforward—using the weakening lemma at variables.

Type soundness — Type substitution

Lemma (Type substititition, 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 require the strengthened version with an arbitrary typing context Γ' on the right. Then, the proof is straightforward.

Compositionality

Lemma (Compositionality)

If $\varnothing \vdash E[M] : \tau$, then there exists τ' such that $\varnothing \vdash M : \tau'$ and all M' verifying $\varnothing \vdash M' : \tau'$ also verify $\varnothing \vdash E[M'] : \tau$.

Remarks

- We need to state compositionality under a context Γ that may at least contain type variables. We allow program variables as well, as it does not complicate the proof.
- Extension of Γ by type variables is needed because evaluation proceeds under type abstractions, hence the evaluation context may need to bind new type variables.

Type soundness — Subject reduction

Theorem (Subject reduction)

Reduction preserves types: if $M_1 \longrightarrow M_2$ then for any context $\vec{\alpha}$ and type τ such that $\vec{\alpha} \vdash M_1 : \tau$, we also have $\vec{\alpha} \vdash M_2 : \tau$.

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 (slide 80).

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 must be been stated using a sequence of type variables $\vec{\alpha}$ for the typing context instead of the empty environment. A closed term does not have free program variables, but may have free type variables (in particular under the value restriction).

The theorem is proved by induction and case analysis on M.

It relies mainly on the *classification lemma* (given below) and the progress assumption for destructors (slide 80).

Type soundness

Classification

Beware! We must take care of partial applications of constants

Lemma (Classification)

Assume $\vec{\alpha} \vdash V : \tau$

- *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.*

  This must be refined by partitioning constructors according to their associated type-constructor:

  If τ is a *G*-constructed type (*e.g.* int, $\tau_1 \times \tau_2$, or $\tau\ list$), then V is a value constructed with a *G*-constructor (*e.g.* an integer n, a pair (V_1, V_2), a list Nil or $Cons(V_1, V_2)$)

Normalization

Theorem

Reduction terminates in pure System F.

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

This is a difficult proof, due to Girard [1972]; Girard et al. [1990]).

See the lesson on logical relations.

Contents

Implicitly-typed System F

The syntax and dynamic semantics of terms are that of the untyped λ-calculus. We use letters a, v, and e to range over implicitly-typed terms, values, and evaluation contexts. We write F and $\lceil F \rceil$ for the explicitly-typed and implicit-typed versions of System F.

Definition 1 A closed term a is in $\lceil F \rceil$ if it is the type erasure of a closed (with respect to term variables) term M in F.

We rewrite the typing rules to operate directly on unannotated terms by dropping all type information in terms:

Definition 2 (equivalent) Typing rules for $\lceil F \rceil$ are those of the implicitlty-typed simply-typed λ-calculus with 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 global assumption on the formation of Γ, α:

$$\text{IF-TABS}\quad \frac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall \alpha . \tau} \qquad\qquad \text{IF-TABS-BIS}\quad \frac{\Gamma \vdash a : \tau \qquad \alpha \# \Gamma}{\Gamma \vdash a : \forall \alpha . \tau}$$

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 as in if-Tabs-Bis:

The side condition is important to avoid unsoundness by violation of the scoping rules.

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

Omitting the side condition leads to *unsoundness:*

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

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

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 :

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

$\alpha_1, \alpha_2, x : \alpha_1, \alpha_1$ 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 $\forall\vec{\alpha}.(\Gamma \Rightarrow \tau)$, where $\vec{\alpha}$ 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.

One could write judgements of the form $\Gamma \vdash a \Rightarrow M : \tau$ to mean that the *explicitly typed* 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? 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 $\lambda fxy.(f\ x, f\ y)$

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.

Neither type is an instance of the other, for any reasonable definition of the word *instance*, because each one has an inhabitant that does not admit the other as a type.

Take, for instance,

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

and

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

Distrib pair in F^ω (parenthesis)

In F^ω, one can abstract over type *functions* (*e.g.* of kind $\star \to \star$) and write:

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

call it "dp" of type:

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

Then

$$\mathsf{dp}\ (\lambda\alpha.\alpha)(\lambda\alpha.\alpha)\\ \quad : \forall\alpha_1.\forall\alpha_2.(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

$$\Lambda\alpha_1.\Lambda\alpha_2.\,\mathsf{d}p\ (\lambda\alpha.\alpha_1)\ (\lambda\alpha.\alpha_2)\ \alpha_1\ \alpha_2\\ \quad : \forall\alpha_1.\forall\alpha_2.(\forall\alpha.\,\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

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 τ_0 of which the two previous 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:

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

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

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

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, τ_1 and τ_2 must be of the form $\forall \vec{\alpha}.\, \tau$ and $\forall \vec{\beta}.\, [\vec{\alpha} \mapsto \vec{\tau}]\tau$ where $\vec{\beta} \# \forall \vec{\alpha}.\, \tau$. *W.l.o.g*, we may assume that $\vec{\beta} \# \Gamma$.

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

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

Admissible rule:

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

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

Retyping contexts in F

In F, subtyping is a judgment $\Gamma \vdash \tau_1 \leq \tau_2$, rather than a binary relation, where the context Γ keeps track of 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 $\Gamma, x : \tau_1 \vdash \mathcal{R}[x] : \tau_2$ (where $x \# \mathcal{R}$)

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

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

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

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

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 having 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 course notes 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.*

Other approaches to type inference in System F

In System F, one can still perform bottom-up type checking, provided type abstractions and type applications are explicit.

One can perform incomplete forms of type inference, such as *local type inference* [Pierce and Turner, 2000; Odersky et al., 2001].

Finally, one can design restrictions or variants of the system that have decidable type inference. Damas and Milner's type system is one example; MLF [Le Botlan and Rémy, 2003] is a more expressive, and more complex, approach.

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 same as the type of 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:

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

Type erasing semantics — Direct simulation

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

Lemma (Direct 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* $\lceil M_1 \rceil \longrightarrow_{\beta\delta} \lceil M_2 \rceil$

Both parts are easy by definition of type erasure.

Type erasing semantics — Inverse simulation

The inverse 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 (Inverse 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 first 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 again and again.
- 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 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 direct 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 inverse simulation in F, there exists M_2 such that $M_1 \longrightarrow_\iota^* \longrightarrow_{\beta\delta} M_2$ 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 versions, with a type-erasing semantics in mind, but this is not always checked in details.

While the direct simulation is usually straightforward, the inverse simulation is often harder. As type systems get 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.

It is also used in program proofs by refinement, proving some properties on a high-level abstract version of a program and using bisimulation to show that the properties also hold for the real concrete version of the program.

Proof of inverse simulation

The inverse 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 F^ω, it requires termination of simply-typed λ-calculus.)

The proof of inverse simulation in the case M is ι-normal is by induction on the reduction in $\lceil F \rceil$, using a few helper lemmas, to deal with the fact that type-erasure is not injective.

Proof of inverse simulation — Helper lemmas

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

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

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

Lemma

1) *A term that erases to $\bar{e}[a]$ can be put in 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 $\Lambda\vec{\alpha}.\,[]\;\vec{\tau}$.*

Proof of inverse simulation

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.

Proof of inverse simulation — 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.* $\square$

Bibliography I

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

▷ Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 54–65, June 2007.

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

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

▷ 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.

Bibliography II

- Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Communications of the ACM*, 8(2):89–101, 1965.
- 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.
- 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 III

- 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.
- 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.
- John C. Reynolds. Three approaches to type structure. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer, March 1985.
- Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000.

Bibliography IV

- Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179(1):1–18, 2002.
- Philip Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, September 1989.
- Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1–3):201–226, May 2007.
- 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 V

▷ Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.