MPRI 2.4

System F

François Pottier

2025–2026

What is a type?

A type is a concise, formal description of the behavior of a program fragment.

For instance, in OCaml, the following are types:

- *int*
  an integer
- $int \rightarrow bool$
  a function that maps an integer argument to a Boolean result
- $(int \rightarrow bool) \rightarrow (int\ list \rightarrow int\ list)$
  a function that maps an integer predicate to an integer list transformer

Sound, static type-checking

Types must be sound: the behavior of a program must obey its type.

- an expression of type *int* must actually produce an integer value, if it terminates.

We want to type-check programs and reject ill-typed programs.

We want to do so at compile time, not at runtime.

Benefits

Types serve as machine-checked documentation.

Types provide a safety guarantee.

> – *Well-typed expressions do not go wrong.*
> Milner, A Theory of Type Polymorphism in Programming, 1978.

Types enable modularity and abstraction,
thereby enabling separate compilation and increasing robustness.

> – *Type structure is a syntactic discipline*
> *for enforcing levels of abstraction.*
> Reynolds, Types, Abstraction and Parametric Polymorphism, 1983.

Types enable potentially greater efficiency.

Types all the way down

Types make sense in low-level programming languages, too:
even assembly language can be typed!

Morrisett et al., From System F to Typed Assembly Language, 1999.

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

- Preserving types helps understand a transformation,
- helps debug it,
- and can pave the way to a semantics preservation proof.

Chlipala, A certified type-preserving compiler from lambda calculus to assembly language, 2007.

Downsides

Types are descriptions of programs,
so annotating programs with types can lead to redundancy.

- There is a need for a certain degree of type inference.

Types restrict expressiveness.

- A sound, decidable type system must reject some safe programs.

Typed or untyped?

Reynolds 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.*
>
> Reynolds, Three Approaches to Type Structure, 1985.

Typed, with ever richer 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.

The simply-typed λ-calculus

Let us first review the simply-typed λ-calculus
and a simple syntactic proof of its type soundness.

For Coq versions of these definitions and proofs, see `STLCDefinition`, `STLCLemmas`, `STLCTypeSoundnessComplete` (and upcoming lecture).

Terms and dynamic semantics

The terms are the pure λ-terms: $t ::= x \mid \lambda x.t \mid t\ t$.

The reduction relation $\cdot \longrightarrow \cdot$ can be defined as follows:

$$
\begin{array}{rcll}
(\lambda x.t)\ v & \longrightarrow & t[v/x] & & (\beta_v) \\
E[t] & \longrightarrow & E[t'] & \text{if } t \longrightarrow t' & (\mathit{context})
\end{array}
$$

where values and evaluation contexts are defined by:

$$
\begin{array}{rcl}
v & ::= & \lambda x.t \\
E & ::= & [\,] \mid E\ t \mid v\ E
\end{array}
$$

Simple types

The syntax of types, in its simplest form, includes type variables and function types:

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

The type system

The type system is a 3-place predicate.

A typing judgement takes the form:

$$\Gamma \vdash t : T$$

A type environment Γ is a finite sequence of bindings of variables to types:

$$\Gamma ::= \emptyset \mid \Gamma; x : T$$

It can also be viewed as a partial function of variables to types.

The type system

The typing judgement is inductively defined:

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x) \qquad \textsc{Abs}\quad \frac{\Gamma; x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \qquad \textsc{App}\quad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

It is syntax-directed.

Stating type soundness

What is a formal statement of Milner's slogan?

– *Well-typed expressions do not go wrong.*
Milner, A Theory of Type Polymorphism in Programming, 1978.

A well-typed, closed program must converge or diverge. It cannot crash.

Theorem (Type Soundness)

If $\emptyset \vdash t : T$ *then either* $\exists v,\ t \longrightarrow^{\star} v \quad$ *or* $\quad t \longrightarrow^{\omega}$.

Establishing type soundness

Type soundness follows from two properties:

Theorem (Subject reduction)

Reduction preserves types:
$\emptyset \vdash t : T$ *and* $t \longrightarrow t'$ *imply* $\emptyset \vdash t' : T$.

Theorem (Progress)

A well-typed, irreducible term is a value:
if $\emptyset \vdash t : T$ *and* $t \not\longrightarrow$ *, then* t *is a value.*

Well-typedness is an invariant that implies absence of crashes.

Wright and Felleisen, A Syntactic Approach to Type Soundness, 1994.

This approach contrasts with a more complex semantic/logical approach:

Timany, Krebbers, Dreyer, Birkedal,
A Logical Approach to Type Soundness, 2024.

Establishing subject reduction

Subject reduction is proved by induction over the hypothesis $t \longrightarrow t'$.

There are two cases, corresponding to (β_v) and $(context)$.

Establishing subject reduction

In the case of (β_v), the first hypothesis is

$$\emptyset \vdash (\lambda x.t)\ v : T_2$$

and the goal is

$$\emptyset \vdash t[v/x] : T_2$$

How do we proceed?

Establishing subject reduction

We decompose the first hypothesis.

Because the type system is syntax-directed, the derivation of the first hypothesis must be of this form, for some type T_1:

$$\textsc{App}\;\dfrac{\textsc{Abs}\;\dfrac{x : T_1 \vdash t : T_2}{\emptyset \vdash \lambda x.t : T_1 \rightarrow T_2} \qquad \emptyset \vdash v : T_1}{\emptyset \vdash (\lambda x.t)\; v : T_2}$$

The goal is still

$$\emptyset \vdash t[v/x] : T_2$$

Where next?

Establishing subject reduction

We need a simple lemma:

Lemma (Value substitution)

Replacing a formal parameter with a type-compatible actual argument preserves types:
$x : T_1 \vdash t : T_2$ *and* $\emptyset \vdash v : T_1$ *imply* $\emptyset \vdash t[v/x] : T_2$.

How do we prove this lemma?

Establishing subject reduction

The lemma must be generalized so it can be proven by induction over a typing judgement for t:

Lemma (Value substitution)

$x : T_1, \Gamma \vdash t : T_2$ *and* $x \notin dom(\Gamma)$ *and* $\emptyset \vdash v : T_1$ *imply* $\Gamma \vdash t[v/x] : T_2$.

The proof is straightforward.

At variables, one must argue that $\emptyset \vdash v : T_1$ implies $\Gamma \vdash v : T_1$ (weakening).

This closes the case of (β_v).

Establishing subject reduction

In the case of rule (*context*), the first hypothesis is

$$\emptyset \vdash E[t] : T$$

The second hypothesis is

$$t \longrightarrow t'$$

where, by the induction hypothesis, this reduction preserves types.

The goal is

$$\emptyset \vdash E[t'] : T$$

How do we proceed?

Establishing subject reduction

Type-checking is compositional. For the judgement $\emptyset \vdash E[t] : T$ to hold, only the type of the subterm in the hole matters, not its exact form.

Lemma (Compositionality)

Assume $\emptyset \vdash E[t] : T$. Then, there exists a type T' such that:

- $\emptyset \vdash t : T'$,
- *for every term t',* $\quad \emptyset \vdash t' : T'$ *implies* $\emptyset \vdash E[t'] : T$.

Using this lemma, the (*context*) case of subject reduction is immediate.

Establishing progress

Recall the statement of Progress:

if $\emptyset \vdash t : T$ *and* $t \not\longrightarrow$ *, then t is a value.*

This can be reformulated in a positive way:

if $\emptyset \vdash t : T$ *then* $t \longrightarrow \cdot$ *or* t *is a value.*

How can we prove this?

Establishing progress

Progress is proved by induction over the term t
or over the hypothesis $\emptyset \vdash t : T$.

Thus, there is one case per construct in the syntax of terms.

In the pure λ-calculus, there are just three cases:

- variable;
- λ-abstraction;
- application.

Two of these are immediate...

Establishing progress

The case of variables cannot occur: a variable is not closed.

The case of λ-abstractions is immediate: a λ-abstraction is a value.

Establishing progress

In the case of applications, the goal is:

if $\emptyset \vdash t_1\ t_2 : T$ *then* $t_1\ t_2 \longrightarrow \cdot$ *or* $t_1\ t_2$ *is a value.*

This goal can be simplified:

if $\emptyset \vdash t_1\ t_2 : T$ *then* $t_1\ t_2 \longrightarrow \cdot$ *.*

Indeed, an application is never a value.

How do we proceed?

Establishing progress

The goal is

if $\emptyset \vdash t_1 \; t_2 : T$ *then* $t_1 \; t_2 \longrightarrow \cdot$.

By inversion of the type-checking rule for applications, we must have $\emptyset \vdash t_1 : T_1 \to T$ and $\emptyset \vdash t_2 : T_1$ for some type T_1.

By the induction hypothesis, t_1 must be reducible or a value v_1.

If t_1 is reducible, then, because $[\,] \; t_2$ is an evaluation context, $t_1 \; t_2$ is reducible as well, and we are done. So, assume t_1 is v_1.

By the induction hypothesis, t_2 must be reducible or a value v_2.

If t_2 is reducible, then, because $v_1 \; [\,]$ is an evaluation context, $v_1 \; t_2$ is reducible as well, and we are done. So, assume t_2 is v_2.

$\emptyset \vdash v_1 : T_1 \to T$ implies that v_1 must be a λ-abstraction (see next slide). So $v_1 \; v_2$ is a β_v-redex: it is reducible. We are done.

Classification of values

We have appealed to the following property:

Lemma (Classification)

Assume $\emptyset \vdash v : T$. Then,

- *if T is an arrow type, then v is a λ-abstraction;*
- *...*

In pure λ-calculus, this result is trivial. In a richer type system, this lemma claims that the head constructor of the type conveys information about the head constructor of the value.

What is polymorphism?

Polymorphism is the possibility that a term may

- simultaneously admit several distinct types
- or be able to operate at several distinct types.

Flavors of polymorphism

Strachey distinguishes

- parametric polymorphism (universal types; today);
- ad hoc polymorphism
  (overloaded operations; see upcoming lecture on type classes).

Strachey, Fundamental Concepts in Programming Languages, 1967.

Why polymorphism?

Polymorphism seems indispensable: a comparison-based sorting function should be applicable to lists of integers, lists of Booleans, etc.

In short, it should have polymorphic type:

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

whose instances are the monomorphic types:

$$\begin{array}{c}(\mathit{int} \rightarrow \mathit{int} \rightarrow \mathit{bool}) \rightarrow \mathit{int}\ \mathit{list} \rightarrow \mathit{int}\ \mathit{list} \\ (\mathit{bool} \rightarrow \mathit{bool} \rightarrow \mathit{bool}) \rightarrow \mathit{bool}\ \mathit{list} \rightarrow \mathit{bool}\ \mathit{list} \\ \ldots\end{array}$$

Why polymorphism?

Without polymorphism, the only ways of achieving this effect would be:

- to duplicate the list sorting function at every type (no-no!);
  - either manually or automatically (C++ templates)
- to claim that the function can sort heterogeneous lists:

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

  In a type system with subtyping, the type $\top$ is the type of all values, and the supertype of all types.

  This leads to a loss of information (bad!):

  - the comparison function must accept two values of any type;
  - sorting a list of (say) integers produces a list of values of unknown type.

  To recover the lost information, a downcast operation is required.

  This approach is common in C and was followed in Java prior to 5.

Polymorphism seems almost free

Some polymorphism is already implicitly present in simply-typed λ-calculus.

The term $\lambda fxy.(f\ x, f\ y)$ admits a principal type:

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

By saying that this term admits a polymorphic type,

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

we make polymorphism internal to the type system.

Towards type abstraction

Polymorphism is one side of type abstraction.

If a sorting function has a polymorphic type:

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

then it knows nothing about X so it must manipulate elements abstractly.

It can move them, copy them, pass them to the comparison function, but cannot directly inspect their structure.

Inside the sorting function, X is an abstract type.

Parametricity

A polymorphic type strongly constrains its inhabitants.

For instance, in a pure and total language, the polymorphic type

$$\forall X.X \to X$$

has only one inhabitant, namely the identity.

Parametricity

Similarly, the type of a polymorphic sorting function:

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

reveals a "free theorem" about its behavior: roughly, the outcome of sorting depends only on the outcomes of comparisons:

> *For all types X_1 and X_2,*
> *for every binary relation R between X_1 and X_2,*
> *if cmp maps related arguments to identical Boolean results,*
> *then sort cmp maps related lists to related lists.*

See the lecture on binary logical relations and parametricity (GS).

System F: types

The polymorphic λ-calculus, also known as System F, was independently defined by Girard (1972) and Reynolds (1974).

Reynolds, Towards a theory of type structure, 1974.

Compared to the simply-typed λ-calculus, the syntax of types is extended with universal types:

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

System F: terms

How should the syntax and semantics of terms be extended?

The function type $T_1 \rightarrow T_2$ has

- an introduction form $\lambda x.t$
- an elimination form $t_1\ t_2$.

So, the universal type $\forall X.T$ should have

- an introduction form $\Lambda X.t$
- an elimination form $t\ T$.

(This is one possible presentation of System F. Others discussed later.)

System F: types and terms

The types include type variables, function types, and universal types:

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

The terms include type abstractions and type applications:

$$t ::= x \mid \lambda x.t \mid t\ t \mid \Lambda X.t \mid t\ T$$

Some authors use abstractions $\lambda x : T.\, t$ where the formal parameter must be annotated with its type. This makes type-checking easier.

For example, the polymorphic identity function is $\Lambda X.\, \lambda x : X.\, x$ and we want it to have the polymorphic type $\forall X.\ X \rightarrow X$.

System F: dynamic semantics

The reduction rules are:

$$
\begin{array}{rcll}
(\lambda x.t)\ v & \longrightarrow & t[v/x] & & (\beta_v) \\
(\Lambda X.t)\ T & \longrightarrow & t[T/X] & & (\iota) \\
E[t] & \longrightarrow & E[t'] & \text{if } t \longrightarrow t' & (\mathit{context})
\end{array}
$$

where values and evaluation contexts are defined by:

$$
\begin{array}{rcl}
v & ::= & \lambda x.t \mid \Lambda X.v \\
E & ::= & [\,] \mid E\ t \mid v\ E \mid \Lambda X.E \mid E\ T
\end{array}
$$

This allows reduction under Λ. (Other choices discussed later.)

System F: type environments

A type environment Γ binds both term variables and type variables:

$$\Gamma ::= \emptyset \mid \Gamma; x : T \mid \Gamma; X$$

A type environment acts as a partial function of variables x to types T. The lookup operation $\Gamma(x)$ is defined on the next slide.

A runtime type environment Δ binds just type variables: $\Delta ::= \emptyset \mid \Delta; X$. This notion is needed because reduction under Λ is permitted.

System F: type environment lookup and hygiene

Lookup in a type environment, a partial function, is defined as follows:

$$\begin{array}{rcll} (\Gamma; x : T)(y) & = & T & \text{if } x = y \\ (\Gamma; x : T)(y) & = & \Gamma(y) & \text{if } x \neq y \\ (\Gamma; X)(x) & = & \Gamma(x) & \text{if } X \# \Gamma(x) \end{array}$$

The condition $X \# \Gamma(x)$ ensures that X does not shadow an older variable by the same name in Γ.

$$\begin{array}{rcl} & X \# T & X \text{ is fresh for } T \\ \text{stands for} & X \notin \mathit{ftv}(T) & X \text{ does not occur free in } T \end{array}$$

The free type variables of a type are defined by

$$\begin{array}{rcl} \mathit{ftv}(X) & = & X \\ \mathit{ftv}(T_1 \to T_2) & = & \mathit{ftv}(T_1) \cup \mathit{ftv}(T_2) \\ \mathit{ftv}(\forall X.T) & = & \mathit{ftv}(T) \setminus \{X\} \end{array}$$

System F: the typing judgement

The typing judgement is inductively defined:

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

$$\textsc{Abs}\quad \frac{\Gamma; x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2}$$

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

$$\textsc{TAbs}\quad \frac{\Gamma; X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T}$$

$$\textsc{TApp}\quad \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t\ T' : T[T'/X]}$$

It is syntax-directed thanks to explicit type abstractions and applications.

Polymorphism is impredicative: a type variable denotes an arbitrary type.

In TAbs, many authors require $X \# \Gamma$. Here, this is not needed because shadowing is prevented by our definition of environment lookup $\Gamma(x)$.

Because a Λ-bound type variable can be renamed, when one uses TAbs to build a type derivation, one can always choose X so that $X \# \Gamma$ holds.

System F in de Bruijn style

With de Bruijn indices, shadowing is avoided by lifting in suitable places.

In this slide, a type environment Γ is a total function of indices to types, or (equivalently) an infinite sequence of types, and environment lookup $\Gamma(x)$ is just function application.

$$\text{JFVar}\quad \Gamma \vdash x : \Gamma(x)$$

$$\text{JFLam}\quad \frac{T_1 \cdot \Gamma \vdash t : T_2}{\Gamma \vdash \lambda t : T_1 \to T_2}$$

$$\text{JFApp}\quad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

$$\text{JFTyAbs}\quad \frac{\Gamma; (+1) \vdash t : T}{\Gamma \vdash \Lambda t : \forall T}$$

$$\text{JFTyApp}\quad \frac{\Gamma \vdash t : \forall T}{\Gamma \vdash t\ T' : T[T' \cdot id]}$$

For an example of this style, see `SystemFDefinition` `(Alectryon)`.

(This Coq file is in Curry style. The above rules are in Church style.)

Establishing type soundness

Type soundness again follows from subject reduction and progress.

Theorem (Subject reduction)

Reduction preserves types:
$\Delta \vdash t : T$ *and* $t \longrightarrow t'$ *imply* $\Delta \vdash t' : T$.

Theorem (Progress)

A well-typed, irreducible term is a value:
if $\Delta \vdash t : T$ *and* $t \not\longrightarrow$ *, then* t *is a value.*

Establishing subject reduction

Subject reduction is still proved by induction over $t \longrightarrow t'$.

As before, there is one case per reduction rule, so now three cases.

- the case of (β_v) is unchanged
  because the type system is syntax-directed.
  A derivation of $\Delta \vdash (\lambda x.t)\ v : T_2$ must use the rules App and Abs.
- the case of (*context*) is unchanged.
- the case of (ι) is new (see next slides).

Establishing subject reduction

In the case of (ι), the first hypothesis is

$$\Delta \vdash (\Lambda X.t)\ T : T_2$$

and the goal is

$$\Delta \vdash t[T/X] : T_2$$

How do we proceed?

Establishing subject reduction

We decompose the first hypothesis.

Because the type system is syntax-directed, the derivation of the first hypothesis must be of this form:

$$\textsc{TApp}\;\dfrac{\textsc{TAbs}\;\dfrac{\Delta; X \vdash t : T_1}{\Delta \vdash \Lambda X.t : \forall X.T_1} \qquad T_1[T/X] = T_2}{\Delta \vdash (\Lambda X.t)\; T : T_2}$$

The goal is still

$$\Delta \vdash t[T/X] : T_2$$

Where next?

Establishing subject reduction

We need a simple lemma:

Lemma (Type substitution)

Replacing a type variable X with an arbitrary type T preserves types:

$$\Delta; X; \Gamma \vdash t : T_1$$

implies

$$\Delta; \Gamma[T/X] \vdash t[T/X] : T_1[T/X]$$

This lemma is the essence of parametric polymorphism.

Its proof is straightforward.

For a statement in de Bruijn style, see `SystemFLemmas` `(Alectryon)`.

Establishing progress

Recall the statement of Progress:

if $\Delta \vdash t : T$ *then* $t \longrightarrow \cdot$ *or* t *is a value.*

As before, progress is proved by induction over the hypothesis $\Delta \vdash t : T$.

There is one case per typing rule:

- the cases of Var, Abs, App are unchanged.
- the cases of TAbs and TApp are new.

Establishing progress

In the case of TAbs, the judgement $\Delta \vdash \Lambda X.t : \forall X.T$
follows from $\Delta; X \vdash t : T$
and the goal is

$$\Lambda X.t \longrightarrow \cdot \quad \textit{or} \quad \Lambda X.t \textit{ is a value.}$$

The induction hypothesis assures us that $t \longrightarrow \cdot$ or t is a value.

- in the first case, the left-hand disjunct of the goal holds, because $\Lambda X.[]$ is an evaluation context.
- in the second case, the right-hand disjunct of the goal holds, because $\Lambda X.v$ is a value.

Establishing progress

In the case of TApp, the judgement $\Delta \vdash t\ T_2 : T[T_2/X]$
follows from $\Delta \vdash t : \forall X.T$
and the goal is

$$t\ T_2 \longrightarrow \cdot \quad \textit{or} \quad t\ T_2 \textit{ is a value.}$$

As $t\ T_2$ is not a value, this goal can be simplified to:

$$t\ T_2 \longrightarrow \cdot \quad .$$

The induction hypothesis assures us that $t \longrightarrow \cdot$ or t is a value.

- in the first case, the goal holds
  because $[\,]\ T_2$ is an evaluation context.
- in the second case, because t is a value and has a universal type, t must be of the form $\Lambda X.v$, so (ι) fires, and the goal holds.

Classification of values

We have again appealed to a classification lemma:

Lemma (Classification)

Assume $\Delta \vdash v : T$. Then,

- *if T is an arrow type, then v is a λ-abstraction;*
- *if T is a universal type, then v is a Λ-abstraction.*

Type erasure

Do type abstractions and applications influence computation?

No.

We have defined $v ::= \ldots \mid \Lambda X.v$ and $E ::= \ldots \mid \Lambda X.E \mid E\ T$.

We intend a type erasure property to hold:

> *The program with or without type abstractions and applications has the same behavior.*

Philosophy of type erasure

Type erasure means that types need not exist at runtime.

Type erasure supports the idea that untyped terms have well-defined behavior and that types are descriptions of pre-existing behavior.

Some researchers disagree. They argue that only typed terms should have a meaning and/or that one should let types influence reduction.

The two views can be reconciled. Instead of letting “types exist at runtime”, one can erase types and use type descriptions (values) at runtime. (See upcoming lecture on GADTs.)

The type erasure property

A typed programming language has the type erasure property if:

$$behaviors(t) = behaviors(\lceil t \rceil)$$

The function $\lceil \cdot \rceil$ erases all type annotations.

$behaviors(t)$ is the set of the observable behaviors of the (closed) term t.

By observable behavior of a (closed, well-typed) term, I mean:

- to converge (to reduce to a value), or
- to diverge (to reduce forever).

(A well-typed term cannot go wrong.)

In pure System F a well-typed term cannot diverge. But one can extend System F with recursive functions; then a well-typed term can diverge.

A type erasure function

The erasure function $\lceil \cdot \rceil$ maps a term to a term:

$$
\begin{array}{rcl}
\lceil x \rceil & = & x \\
\lceil \lambda x.t \rceil & = & \lambda x.\lceil t \rceil \\
\lceil t_1 \; t_2 \rceil & = & \lceil t_1 \rceil \, \lceil t_2 \rceil \\
\lceil \Lambda X.t \rceil & = & \lceil t \rceil \\
\lceil t \; T \rceil & = & \lceil t \rceil
\end{array}
$$

Simulation

To prove that type erasure holds, we wish to show that computing with type annotations is "the same" as computing without them.

To do so, one approach is to prove a (forward) simulation statement.

Roughly,

> *If one step of computation with type annotations can be made, then one step of computation without them can be made.*

Simulation, take 1

Here is a first attempt at a simulation statement:

Lemma (Simulation)

If $t \longrightarrow t'$ then $\lceil t \rceil \longrightarrow \lceil t' \rceil$.

Is this true?

No. We must allow stuttering, that is, zero steps on the right-hand side.

Simulation, take 2

Here is a second simulation statement:

Lemma (Simulation)
If $t \longrightarrow t'$ then $\lceil t \rceil \longrightarrow^{?} \lceil t' \rceil$.

We write $\longrightarrow^{?}$ for zero or one step along the reduction relation $\longrightarrow$.

Is this true?

Yes. The proof is by induction over $t \longrightarrow t'$. (Exercise: do it!)

Is this enough?

Are we happy with (just) this simulation statement?

Does it really mean that t and $\lceil t \rceil$ compute "the same thing"?

If we had posited $\lceil t \rceil \triangleq \lambda x.x$ then it would still hold!

Initially I wanted to prove that $\lceil \cdot \rceil$ preserves observable behavior:

- if t converges then $\lceil t \rceil$ converges;
- if t diverges then $\lceil t \rceil$ diverges.

Preservation of convergence

We must check:

Lemma (Erasure of a value)
For every value v, $\lceil v \rceil$ is a value.

Recall the definition of values: $v ::= \lambda x.t \mid \Lambda X.v$.

The proof (by induction on v) is easy.

From this and from the Simulation statement one can deduce that if t converges then $\lceil t \rceil$ converges.

Preservation of divergence

We must check:

Lemma (Erasure of a divergent computation)
If t diverges then $\lceil t \rceil$ diverges.

Is this true?

Recall the statement of Simulation: *if $t \longrightarrow t'$ then $\lceil t \rceil \longrightarrow^{?} \lceil t' \rceil$.*

This does not allow proving that $t \longrightarrow^{\omega}$ implies $\lceil t \rceil \longrightarrow^{\omega}$.

We must find a way of proving that $\lceil t \rceil$ cannot stutter forever.

How?

Simulation, take 3

Here is a final simulation statement:

Lemma (Simulation)

If $t \longrightarrow t'$ then

- *either $\lceil t \rceil \longrightarrow \lceil t' \rceil$*
- *or $\lceil t \rceil = \lceil t' \rceil$ and $size(t) > size(t')$*

where size maps terms into $\mathbb{N}$.

Exercise: define *size* and do the proof (by induction over $t \longrightarrow t'$).

Preservation of divergence

Now one can prove:

Lemma (Erasure of a divergent computation)
If t diverges then $\lceil t \rceil$ diverges.

Preservation of going-wrongness

Can we prove this?

Lemma

If t goes wrong then $\lceil t \rceil$ goes wrong.

No. This statement is false.

$(\Lambda X.\lambda x.x)\ 0$ is stuck, yet its erasure $(\lambda x.x)\ 0$ is not stuck.

We won't need this statement anyway
because we care about well-typed terms only.

Preservation of observable behavior

Let $\downarrow$, $\nearrow$, and $\lightning$ stand for the three possible observable behaviors: convergence, divergence, and going wrong.

Let *behaviors*(t) stand for the set

$$
\begin{array}{llll}
\{ \; \downarrow & \mid & \exists v, t \longrightarrow^\star v & \} \cup \\
\{ \; \nearrow & \mid & t \longrightarrow^\omega & \} \cup \\
\{ \; \lightning & \mid & \exists t', t \longrightarrow^\star t' \wedge t' \text{ is stuck} & \}
\end{array}
$$

By putting together the previous results, we get:

Lemma (Forward preservation of observable behavior)
if t cannot go wrong then behaviors$(t) \subseteq$ behaviors$(\lceil t \rceil)$.

Preservation of observable behavior

We have just proved:

Lemma (Forward preservation of observable behavior)
if t cannot go wrong then $behaviors(t) \subseteq behaviors(\lceil t \rceil)$.

Because erased terms have deterministic semantics,
$behaviors(\lceil t \rceil)$ must be a singleton set.

Because every term has some behavior,
$behaviors(t)$ must be a nonempty set.

Thus, we have:

Lemma (Preservation of observable behavior)
if t cannot go wrong then $behaviors(t) = behaviors(\lceil t \rceil)$.

Corollary (Preservation of safety)
if t cannot go wrong then $\lceil t \rceil$ cannot go wrong.

Type erasure

We have just proved a type erasure property.

One can

- use type-annotated terms
  when defining the type discipline and proving type soundness,
- erase type annotations when executing terms,
- and all will be well,
  provided ill-typed terms are rejected up front.

Type erasure without determinism

What if the semantics is not deterministic?

A backward simulation statement seems necessary:

Lemma

If $\lceil t \rceil = u$ and $u \longrightarrow u'$ and $\emptyset \vdash t : T$ then there exists t' such that $t \longrightarrow t'$ and

- *either $\lceil t' \rceil = u'$*
- *or $\lceil t' \rceil = u$ and $size(t) > size(t')$.*

From this, we get:

Lemma (Backward preservation of observable behavior)

If $\emptyset \vdash t : T$ then $behaviors(\lceil t \rceil) \subseteq behaviors(t)$.

Thus, if t is well-typed then t and $\lceil t \rceil$ have the same behaviors.

Caveat

I have not formalized the proofs about erasure in Coq.

I would need terms that contain term binders λ and type binders Λ, which AutoSubst 1 does not support.

AutoSubst 2 supports this, but I have not tried it.

Variants of System F

Letting type abstractions and applications appear in the syntax of terms is known as Church style.

- We have studied this style, in a variant where reduction under Λ is permitted, and we have proved type erasure.
- There is also a variant where $\Lambda X.t$ is a value and reduction under Λ is not permitted. Our results about type erasure can be preserved provided $\lceil \Lambda X.t \rceil$ is defined as $\lambda().\lceil t \rceil$ and $\lceil t\ T \rceil$ is $\lceil t \rceil\ ()$.

It is also possible to not mark type abstractions and applications in the syntax of terms: this is known as Curry style.

- In this style, there is no need to prove type erasure.
- A proof of type soundness in this style is possible but more difficult. In particular, the classification lemma becomes more involved. See `SystemFTypeSoundnessComplete` `(Alectryon)`.

System F in Curry style

This is System F with implicit type abstractions and applications:

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

$$\textsc{Abs}\quad \frac{\Gamma; x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2}$$

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

$$\textsc{TAbs}\quad \frac{\Gamma; X \vdash t : T}{\Gamma \vdash t : \forall X.T}$$

$$\textsc{TApp}\quad \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t : T[T'/X]}$$

The rules TAbs and TApp are not syntax-directed.

System F in Curry style

And here is an equivalent presentation with a subtyping rule Sub:

$$\text{Var}\quad \Gamma \vdash x : \Gamma(x) \qquad \text{Abs}\quad \frac{\Gamma; x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \qquad \text{App}\quad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

$$\text{TAbs}\quad \frac{\Gamma; X \vdash t : T}{\Gamma \vdash t : \forall X.T} \qquad \text{Sub}\quad \frac{\Gamma \vdash t : T \qquad T \leq T'}{\Gamma \vdash t : T'}$$

where $T \leq T'$ is defined on the next slide.

Subtyping in System F

Subtyping is defined by

$$\textsc{Inst}\quad \forall X.T \leq T[T'/X] \qquad\qquad \textsc{Gen}\quad \frac{X \# T}{T \leq \forall X.T} \qquad\qquad \textsc{Transitivity}\quad \frac{T_1 \leq T_2 \qquad T_2 \leq T_3}{T_1 \leq T_3}$$

Exercise: check that these two presentations of System F in Curry style are indeed equivalent.

A richer notion of subtyping: System F_η

Mitchell (1988) defines System F_η, a more powerful variant of System F, based on a richer subtyping relation:

Inst
$$\forall X.T \leq T[T'/X]$$

Gen
$$\frac{X \mathrel{\#} T}{T \leq \forall X.T}$$

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

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

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

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

Clearly $\Gamma \vdash_F t : T$ implies $\Gamma \vdash_{F_\eta} t : T$.

Conversely $\Gamma \vdash_{F_\eta} t : T$ implies $\Gamma \vdash_F t' : T$ for some t' such that $t \equiv_\eta t'$.

Exercise: prove this claim!

Therefore System F_η is the closure of System F under η-equality.

The type inference problem

Let u be a (closed) unannotated term.

Provided we decorate it with type abstractions and applications, is it well-typed? Can one find a type for it?

Can one find t and T such that $\lceil t \rceil = u$ and $\emptyset \vdash t : T$?

Wells (1999) proves that this problem is undecidable.

An example

Consider the unannotated term $\lambda fxy.(f\ x, f\ y)$.

Here is one way of decorating it:

$$\Lambda X_1.\Lambda X_2.\lambda f\colon X_1 \to X_2.\lambda x\colon X_1.\lambda y\colon X_1.(f\ x, f\ y)$$

For readability, we have also annotated every λ binder with its type.

This term admits the polymorphic type:

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

This is the type that would be inferred by OCaml.

An example

This untyped term can also be decorated in a different way:

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

This term admits the polymorphic type:

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

This begs a question...

Incomparable types in System F

Is one of these two types "more general" than the other?
And if so, in what sense?

$$\begin{array}{l} \forall X_1.\forall X_2.\ \ (X_1 \to X_2)\ \ \to X_1 \to X_1 \to X_2 \times X_2 \\ \forall X_1.\forall X_2.\ (\forall X.X \to X) \to X_1 \to X_2 \to X_1 \times X_2 \end{array}$$

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

Neither can be "more general than" the other,
for any reasonable definition of the relation "more general than",
because each has an inhabitant that does not inhabit the other.

Exercise: find these inhabitants!

Absence of principal types

I believe that the unannotated term $\lambda fxy.(f\ x, f\ y)$ does not admit a type that is more general than the previous two types according to the subtyping relation of System F.

In other words, System F does not have principal types.

Type inference in System F_η

One might hope that type inference is easier in System F_η than in System F.

Unfortunately Tiuryn and Urzyczyn (1995) prove that in System F_η even just the subtyping problem is undecidable.

This implies that typability is System F_η is undecidable (Wells, 1996).

Chrzaszcz (1998) proves that even without DISTRIBUTIVITY the subtyping problem of System F_η is undecidable.

Strong normalization

Strong reduction allows β-reduction everywhere, including under λ and Λ.

Theorem (Strong normalization)

If $\Gamma \vdash t : T$ then every strong reduction sequence out of t is finite.

This result, due to Girard (1972), is more accessibly described in the textbook Proofs and Types by Girard, Lafont and Taylor (1990).

The proof uses logical relations (see upcoming lecture by GS).

Logical consistency

Through the Curry-Howard isomorphism, System F is also a logic, known as second-order logic.

$\emptyset \vdash t : T$ means that t is a proof of the proposition T.

Strong normalization implies that second-order logic is consistent: there is no proof of $\forall X.X$.

- If there was a closed term of type $\forall X.X$,
- then (by strong normalization) this term would reduce to a value
- and (by subject reduction) this (closed) value would have type $\forall X.X$
- but (by classification of values) a closed value cannot have this type.