

MPRI 2.4

System F

François Pottier



2024

What is a type?

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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.

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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.

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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.

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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:

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

where values and evaluation contexts are defined by:

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

Simple types

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

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

The type system

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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:

$$\begin{array}{c} \text{VAR} \\ \Gamma \vdash x : \Gamma(x) \end{array} \qquad \begin{array}{c} \text{ABS} \\ \dfrac{\Gamma; x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x. t : T_1 \rightarrow T_2} \end{array} \qquad \begin{array}{c} \text{APP} \\ \dfrac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2} \end{array}$$

It is **syntax-directed**.

Stating type soundness

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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 $\exists v, t \longrightarrow^ v$ or $t \longrightarrow^\omega$.*

Establishing type soundness

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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 $t \not\rightarrow$, 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.

Establishing subject reduction

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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 :

$$\text{APP} \frac{\text{ABS} \frac{x : T_1 \vdash t : T_2}{\emptyset \vdash \lambda x. t : T_1 \rightarrow T_2} \quad \emptyset \vdash v : T_1}{\emptyset \vdash (\lambda x. t) v : T_2}$$

The goal is still

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

Where next?

Establishing subject reduction

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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 $x \notin \text{dom}(\Gamma)$ 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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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' , $\emptyset \vdash t' : T'$ 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 $t \not\rightarrow$, then t is a value.

This can be reformulated in a positive way:

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

How can we prove this?

Establishing progress

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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 $\emptyset \vdash t_1 : T_1 \rightarrow T$ 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 .

$\emptyset \vdash v_1 : T_1 \rightarrow T$ 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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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
(e.g., overloaded arithmetic operations; upcoming lecture by PED).

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:

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

whose **instances** are the monomorphic types:

$$\begin{aligned} & (\text{int} \rightarrow \text{int} \rightarrow \text{bool}) \rightarrow \text{int list} \rightarrow \text{int list} \\ & (\text{bool} \rightarrow \text{bool} \rightarrow \text{bool}) \rightarrow \text{bool list} \rightarrow \text{bool list} \\ & \dots \end{aligned}$$

Why polymorphism?

Without 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 can sort **heterogeneous** lists:

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

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

This leads to a **loss of information**. To recover this information, a **downcast** operation is required.

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

Polymorphism seems almost free

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

Polymorphism is one side of **type abstraction**.

If a sorting function has a polymorphic type:

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

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**.

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

A polymorphic type strongly constrains its inhabitants.

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

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

has **only one inhabitant**, namely the identity.

Similarly, the type of a polymorphic sorting function:

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

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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**:

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

System F : terms

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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.

System F: dynamic semantics

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

The reduction rules are:

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

where values and evaluation contexts are defined by:

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

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

System F : type environments

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

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

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

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

The **free type variables** of a type are defined by

$$\begin{aligned}\text{ftv}(X) &= X \\ \text{ftv}(T_1 \rightarrow T_2) &= \text{ftv}(T_1) \cup \text{ftv}(T_2) \\ \text{ftv}(\forall X. T) &= \text{ftv}(T) \setminus \{X\}\end{aligned}$$

System F: the typing judgement

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

The typing judgement is inductively defined:

$$\begin{array}{c}
 \text{VAR} \\
 \Gamma \vdash x : \Gamma(x)
 \end{array}
 \qquad
 \begin{array}{c}
 \text{ABS} \\
 \dfrac{\Gamma; x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x. t : T_1 \rightarrow T_2}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{APP} \\
 \dfrac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}
 \end{array}$$

$$\begin{array}{c}
 \text{TAbs} \\
 \dfrac{\Gamma; X \vdash t : T}{\Gamma \vdash \Lambda X. t : \forall X. T}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{TApp} \\
 \dfrac{\Gamma \vdash t : \forall X. T}{\Gamma \vdash t T' : T[T'/X]}
 \end{array}$$

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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.

$$\begin{array}{c}
 \text{JFVAR} \\
 \Gamma \vdash x : \Gamma(x)
 \end{array}
 \qquad
 \begin{array}{c}
 \text{JFLAM} \\
 \frac{\Gamma_1 \cdot \Gamma \vdash t : T_2}{\Gamma \vdash \lambda t : T_1 \rightarrow T_2}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{JFAPP} \\
 \frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 \ t_2 : T_2}
 \end{array}$$

$$\begin{array}{c}
 \text{JFTYABS} \\
 \frac{\Gamma; (+1) \vdash t : T}{\Gamma \vdash \lambda t : \forall T}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{JFTYAPP} \\
 \frac{\Gamma \vdash t : \forall T}{\Gamma \vdash t \ T' : T[T' \cdot id]}
 \end{array}$$

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

(Warning: this Coq file is in Curry style.)

Establishing type soundness

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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 $t \not\rightarrow$, then t is a value.

Establishing subject reduction

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

We **decompose** the first hypothesis.

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

$$\text{TAbs} \frac{\Delta; X \vdash t : T_1}{\Delta \vdash \lambda X. t : \forall X. T_1} \quad T_1[T/X] = T_2$$

$$\text{TApp} \frac{\quad}{\Delta \vdash (\lambda X. t) T : T_2}$$

The goal is still

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

Where next?

Establishing subject reduction

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

$\lambda X.t \longrightarrow \cdot$ or $\lambda X.t$ 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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

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

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.*

Do type abstractions and applications influence computation?

No.

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

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:

$$\text{behaviors}(t) = \text{behaviors}(\lceil t \rceil)$$

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

$\text{behaviors}(t)$ is the set of the observable behaviors of the (closed) term t .

A type erasure function

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

The **erasure** function $[\cdot]$ maps a term to a term:

$$\begin{aligned} [x] &= x \\ [\lambda x. t] &= \lambda x. [t] \\ [t_1 \ t_2] &= [t_1] [t_2] \\ [\Lambda X. t] &= [t] \\ [t \ T] &= [t] \end{aligned}$$

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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 $\llbracket t \rrbracket \longrightarrow^? \llbracket t' \rrbracket$.

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 $\llbracket t \rrbracket$ compute “the same thing”?

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

We must also show that $\llbracket \cdot \rrbracket$ preserves the **observable behavior** of a term.

The three possible observable behaviors of a (closed) term are:

- to **converge** (to reduce to a value),
- to **diverge** (to reduce forever),
- and to **go wrong** (to reduce to a stuck term).

Preservation of values

We must check:

Lemma (Erasure of a value)

For every value v , $\llbracket v \rrbracket$ is a value.

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

The proof (by induction on v) is easy.

Preservation of divergence

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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?

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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 $\text{size}(t) > \text{size}(t')$*

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

Can we prove this?

Lemma

If t goes wrong then $\llbracket t \rrbracket$ 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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

Let \Downarrow , \nearrow , and $\not\downarrow$ stand for the three possible observable behaviors: convergence, divergence, and going wrong.

Let $behaviors(t)$ stand for the set

$$\begin{aligned} & \{ \Downarrow \mid \exists v, t \longrightarrow^* v \} \cup \\ & \{ \nearrow \mid t \longrightarrow^{\omega} \} \cup \\ & \{ \not\downarrow \mid \exists t', t \longrightarrow^* t' \wedge t' \text{ is stuck} \} \end{aligned}$$

By putting together the previous results, we get:

Lemma (Forward preservation of observable behavior)

if t cannot go wrong then $behaviors(t) \subseteq behaviors(\llbracket t \rrbracket)$.

Preservation of observable behavior

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

We have just proved:

Lemma (Forward preservation of observable behavior)

if t cannot go wrong then $\text{behaviors}(t) \subseteq \text{behaviors}(\lceil t \rceil)$.

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

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

Thus, we have:

Lemma (Preservation of observable behavior)

if t cannot go wrong then $\text{behaviors}(t) = \text{behaviors}(\lceil t \rceil)$.

Corollary (Preservation of safety)

if t cannot go wrong then $\lceil t \rceil$ cannot go wrong.

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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 $\text{size}(t) > \text{size}(t')$.*

From this, we get:

Lemma (Backward preservation of observable behavior)

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

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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. Then type erasure does **not** hold: for instance, $\Lambda X.\omega$ is a value but its erasure ω diverges.

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

This is System F with **implicit** type abstractions and applications:

$$\begin{array}{c}
 \text{VAR} \\
 \Gamma \vdash x : \Gamma(x)
 \end{array}
 \qquad
 \begin{array}{c}
 \text{ABS} \\
 \dfrac{\Gamma; x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x. t : T_1 \rightarrow T_2}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{APP} \\
 \dfrac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}
 \end{array}$$

$$\begin{array}{c}
 \text{TAbs} \\
 \dfrac{\Gamma; X \vdash t : T}{\Gamma \vdash t : \forall X. T}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{TApp} \\
 \dfrac{\Gamma \vdash t : \forall X. T}{\Gamma \vdash t : T[T'/X]}
 \end{array}$$

The rules TAbs and TApp are not syntax-directed.

System F in Curry style

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

And here is an **equivalent** presentation with a **subtyping** rule **SUB**:

$$\begin{array}{c}
 \text{VAR} \\
 \Gamma \vdash x : \Gamma(x)
 \end{array}
 \qquad
 \begin{array}{c}
 \text{ABS} \\
 \dfrac{\Gamma; x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x. t : T_1 \rightarrow T_2}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{APP} \\
 \dfrac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}
 \end{array}$$

$$\begin{array}{c}
 \text{TAbs} \\
 \dfrac{\Gamma; X \vdash t : T}{\Gamma \vdash t : \forall X. T}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{SUB} \\
 \dfrac{\Gamma \vdash t : T \quad T \leq T'}{\Gamma \vdash t : T'}
 \end{array}$$

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

Subtyping in System F

Subtyping is defined by

$$\text{INST} \quad \frac{}{\forall X. T \leq T[T'/X]}$$

$$\text{GEN} \quad \frac{X \# T}{T \leq \forall X. T}$$

$$\text{TRANSITIVITY} \quad \frac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3}$$

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

A richer notion of subtyping: System F_η

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

$$\begin{array}{c}
 \text{INST} \\
 \forall X. T \leq T[T'/X]
 \end{array}
 \qquad
 \begin{array}{c}
 \text{GEN} \\
 \frac{X \# T}{T \leq \forall X. T}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{TRANSITIVITY} \\
 \frac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3}
 \end{array}$$

$$\begin{array}{c}
 \text{DISTRIBUTIVITY} \\
 \frac{\forall \bar{X}. (T_1 \rightarrow T_2)}{\leq (\forall \bar{X}. T_1) \rightarrow (\forall \bar{X}. T_2)}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{CONGRUENCE} \rightarrow \\
 \frac{T_2 \leq T_1 \quad T'_1 \leq T'_2}{T_1 \rightarrow T'_1 \leq T_2 \rightarrow T'_2}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{CONGRUENCE} \forall \\
 \frac{T_1 \leq T_2}{\forall X. T_1 \leq \forall X. T_2}
 \end{array}$$

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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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 $\llbracket t \rrbracket = u$ and $\emptyset \vdash t : T$?

Wells (1999) proves that this problem is undecidable.

An example

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

Here is one way of decorating it:

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

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

This term admits the polymorphic type:

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

This is the type that would be **inferred** by OCaml.

An example

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

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

This term admits the polymorphic type:

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

This begs a question...

Incomparable types in System F

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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

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

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

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

Neither 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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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.

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

But, to clarify this statement, I should define when T_1 is **more general** than T_2 , which we usually write $T_1 \leq T_2$.

We also say that T_2 is an **instance** of T_1 .

Type inference in System F_η

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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 in 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

Types

Type safety

Polymorphism

System F

Type erasure

Digression

Variants

Type inference

Normalization

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.