

# MPRI 2.4

## System F

François Pottier



2023

# 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

## Type safety

## Polymorphism

## Erasure

## System F

## Trivia

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

#### Erasure

#### System F

#### Trivia

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 $\lambda$ -calculus

Let us first review the simply-typed  $\lambda$ -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  $\lambda$ -terms:  $t ::= x \mid \lambda x.t \mid t \ t$ .

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

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

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

Types

Type safety

Polymorphism

Erasure

System F

Trivial

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

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

# The type system

The type system is a 3-place predicate.

A **typing judgement** takes the form:

$$\Gamma \vdash t : T$$

A **type environment**  $\Gamma$  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

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

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.

## Establishing subject reduction

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

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



## Establishing subject reduction

In the case of  $(\beta_v)$ , 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  $T_1$ :

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

The goal is still

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

Where next?

## Establishing subject reduction

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

## 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  $t'$ ,  $\emptyset \vdash t' : T'$  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

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  $\lambda$ -calculus, there are just three cases:

- variable;
- $\lambda$ -abstraction;
- application.

Two of these are immediate...



## Establishing progress

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

The case of  $\lambda$ -abstractions is immediate: **a  $\lambda$ -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

Types

Type safety

Polymorphism

Erasure

System F

Trivial

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  $\lambda$ -abstraction (see next slide). So  $v_1 v_2$  is a  $\beta_v$ -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  $\lambda$ -abstraction;
- ...

In pure  $\lambda$ -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

Some polymorphism is already implicitly present in simply-typed  $\lambda$ -calculus.

The term  $\lambda f x y. (f\ x, f\ y)$  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 a step on the road towards **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**.

## Parametricity

Types

Type safety

Polymorphism

Erasure

System F

Trivia

A type can reveal a lot of information about the terms that inhabit it.  
For instance, in a pure and total language, the polymorphic type

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

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

## Parametricity

Types

Type safety

Polymorphism

Erasure

System F

Trivia

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  $\text{cmp}$  maps related arguments to identical Boolean results,  
then  $\text{sort cmp}$  maps related lists to related lists.*

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

## System F: types

The polymorphic  $\lambda$ -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  $\lambda$ -calculus, the syntax of **types** is extended with **universal types**:

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

## System F: terms

How should the syntax and semantics of **terms** be extended?

Several answers are possible:

- **not at all!** “Curry style”. Keep type information implicit.  
*Types do not exist at runtime and do not influence computation.*
- with **type abstractions** and **type applications**:  $t ::= \dots \mid \Lambda X.t \mid t\ T$   
and with a new reduction rule:

$$(\Lambda X.t)\ T \longrightarrow t[T/X]$$

“Church style”. explicitly show where universal quantifiers are introduced and eliminated.

Does “Church style” imply that type abstractions and applications necessarily influence computation?

## Type erasure

Must type abstractions and applications influence computation?

Several answers are again possible:

- **No.** Define  $v ::= \dots \mid \Lambda X.v$  and  $E ::= \dots \mid \Lambda X.E \mid E T$ .  
Then a **type erasure** property holds: the program **with or without** type abstractions and applications has the same behavior.  
*Types need not exist at runtime and do not influence computation.*
- **Yes.** Define  $v ::= \dots \mid \Lambda X.t$  and  $E ::= \dots \mid E T$ .  
Then type erasure does not hold:  
e.g.,  $\Lambda X.\omega$  is a value but its erasure  $\omega$  diverges.

## The type erasure property

A typed programming language has **the type erasure property** if:

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

where the function  $\lceil \cdot \rceil$  erases all type annotations and  $\text{behaviors}(t)$  is the set of the observable behaviors of the (closed) term  $t$ .



## 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 post hoc 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.)

## What style?

A formal definition of a typed language must choose:

- between **implicit** or **explicit** type information in terms;
  - Here, limited technical impact; mainly a matter of stylistic preference.
  - In richer calculi (dependent types), this can make a big difference.
- whether **type erasure** is desired.
  - I like the idea that types need not exist at runtime.

For a presentation of System  $F$  in implicit style, see upcoming lecture on Syntactic type soundness for System F in Coq.

Today, I use a presentation in **explicit** style, **with** a type erasure property.

System  $F$  in explicit style

The syntax and operational semantics are as follows:

$t$	$::=$	$\dots \mid \Lambda X.t \mid t \ T$	terms
$v$	$::=$	$\dots \mid \Lambda X.v$	values
$E$	$::=$	$\dots \mid \Lambda X.E \mid E \ T$	evaluation contexts
		$(\Lambda X.t) \ T \longrightarrow t[T/X]$	a new reduction rule

Reduction is non-deterministic. This will not be a problem.

Before going further, let us check that the [type erasure](#) property holds.

## A type erasure function

The **erasure** function  $\llbracket \cdot \rrbracket$  maps a term to a term:

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

What statement(s) do we wish to prove about this function?

We wish to state that computing **with type annotations** is “the same” as computing **without them**.

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

Is this true?

**Yes.** The proof is by induction over  $t \longrightarrow t'$ .

## 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!

We must also show that  $\lceil \cdot \rceil$  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

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

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

The proof is by induction over  $t \longrightarrow t'$ .

## Preservation of divergence

### Lemma

*If  $t \longrightarrow^\omega$  diverges then  $\llbracket t \rrbracket \longrightarrow \cdot$ .*

*More precisely, there exists  $t'$  such that  $t'$  diverges and  $\llbracket t \rrbracket \longrightarrow \llbracket t' \rrbracket$ .*

### Proof.

If  $\llbracket t \rrbracket$  could not make one step then (by Simulation) we would have an infinite sequence  $t_1 \longrightarrow t_2 \longrightarrow \dots$  with  $\text{size}(t_1) > \text{size}(t_2) > \dots$ . □

### Lemma (Erasure of a divergent computation)

*If  $t$  diverges then  $\llbracket t \rrbracket$  diverges.*

### Proof.

By infinite iteration of the previous lemma. □

# Preservation of going-wrongness

Types

Type safety

Polymorphism

Erasure

System F

Trivia

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

Erasure

System F

Trivial

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

Let  $\text{behaviors}(t)$  stand for the set

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

By putting together the previous results, we get:

**Lemma (Forward preservation of observable behavior)**

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

## Preservation of observable behavior

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.*

## 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

Types

Type safety

Polymorphism

Erasure

System F

Trivial

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.



## Caveat

I have not formalized the proofs about erasure in Coq.

I would need terms that contain term binders  $\lambda$  and type binders  $\Lambda$ , which AutoSubst 1 does not support.

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

System  $F$ : terms

The **terms** include **type abstractions** and **type applications**:

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

## System F: dynamic semantics

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:

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

## System F: types and type environments

Types

Type safety

Polymorphism

Erasure

System F

Trivia

The **types** include **universal types**:

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

Type environments bind both term variables and **type variables**:

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

They still act as partial functions of variables  $x$  to types  $T$ .

A “runtime type environment” binds just type variables:  $\Delta ::= \emptyset \mid \Delta; X$ .  
This notion is used in some lemmas.

## Hygiene

Types

Type safety

Polymorphism

Erasure

System F

Trivia

Whenever  $\Gamma; X$  is constructed, e.g. in the typing rule  $\text{TAbs}$ , we require  $X \# T$  (“ $X$  is fresh for  $T$ ”), which stands for  $X \notin \text{ftv}(T)$  (“ $X$  does not occur free in  $T$ ”).

This ensures that  $X$  **does not shadow** older variables in  $\Gamma$ .

Furthermore, we allow **renaming** type variables bound by  $\Lambda$  or  $\forall$ .

These statements make sense when type variables are **named**. If type variables are **de Bruijn indices** then shadowing is avoided by **lifting** types in suitable places: see **SystemFDefinitions**.

## System F: the typing judgement

The typing judgement is inductively defined:

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

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

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

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

## 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  $t \not\rightarrow$ , 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  $(\beta_v)$  is unchanged  
because the type system is syntax-directed.

A derivation of  $\Delta \vdash (\lambda x.t) v : T_2$  must use the rules  $A_{PP}$  and  $A_{BS}$ .

- the case of  $(context)$  is unchanged.
- the case of  $(\iota)$  is new (see next slides).

In a Curry-style presentation, the case of  $(\iota)$  would not exist at all, but the case of  $(\beta_v)$  would be more complex: see [SystemFTypeSoundnessComplete](#).



## Establishing subject reduction

In the case of  $(\iota)$ , 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:

$$\text{TA}_{\text{APP}} \frac{\text{TA}_{\text{ABS}} \frac{\Delta; X \vdash t : T_1 \quad X \# \Delta}{\Delta \vdash \lambda X. t : \forall X. T_1} \quad T_1[T/X] = T_2}{\Delta \vdash (\lambda X. t) T : T_2}$$

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:*  
 $\Delta; X; \Gamma \vdash t : T_1$  and  $X \notin \text{btv}(\Gamma)$  imply  $\Delta; \Gamma \vdash t[T/X] : T_1[T/X]$ .

This lemma is [the essence of parametric polymorphism](#).

Its proof is straightforward.

## 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  $\text{TAbs}$ , 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

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

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

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

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

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  $(\iota)$  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  $\lambda$ -abstraction;
- if  $T$  is a universal type, then  $v$  is a  $\Lambda$ -abstraction.

This lemma remains simple because we use a syntax-directed presentation of System  $F$ .

In Curry style, a more complex inversion lemma is needed:  
see [SystemFTypeSoundnessComplete](#).

## Strong normalization

**Strong reduction** allows  $\beta$ -reduction everywhere, including under  $\lambda$  and  $\Lambda$ .

### 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).

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  $\forall X.X$ .



## 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  $\llbracket t \rrbracket = u$  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:

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

For readability, we have also annotated every  $\lambda$  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

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

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

$$\begin{aligned} & \forall X_1. \forall X_2. (X_1 \rightarrow X_2) \rightarrow X_1 \rightarrow X_1 \rightarrow X_2 \times X_2 \\ & \forall X_1. \forall X_2. (\forall X. X \rightarrow X) \rightarrow X_1 \rightarrow X_2 \rightarrow X_1 \times X_2 \end{aligned}$$

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.

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  $T_1$ .

System  $F$  in Curry style

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

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

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

The rules  $\text{TAbs}$  and  $\text{TApp}$  are not syntax-directed.

## System F in Curry style

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

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

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

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  $F_\eta$ 

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

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

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

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  $F_\eta$  is the closure of System  $F$  under  $\eta$ -equality.

## Type inference in System $F_\eta$

One might hope that type inference is easier in System  $F_\eta$  than in System  $F$ .

Unfortunately **Tiuryn and Urzyczyn (1995)** prove that in System  $F_\eta$  even just **the subtyping problem is undecidable**.

This implies that typability in System  $F_\eta$  is undecidable (Wells, 1996).

**Chrzaszcz (1998)** proves that **even without DISTRIBUTIVITY** the subtyping problem of System  $F_\eta$  is undecidable.