2-4-2 / Type systems
Simple types

François Pottier

September 15 and 22, 2009

INRIA

Contents

These slides (as well as other material and information) are available at http://gallium.inria.fr/~fpottier/mpri/.

On functional programming

Functional programming views functions as the primary kind of data, and discourages the use of modifiable data.

Functional programming languages are traditionally *typed* (Scheme and Erlang are exceptions) and have close connections with logic.

Functional programming languages differ in that some are *strict* (ML) and some are *lazy* (Haskell) [Hughes, 1989]. This difference has little impact on typing.

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

For instance, the following are ML types:

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

Types serve as *machine-checked* documentation.

Types provide a *safety* guarantee.

> *"Well-typed expressions do not go wrong."* [Milner, 1978]

Types encourage *separate compilation*, *modularity*, and *abstraction*.

> *"Type structure is a syntactic discipline for enforcing levels of abstraction."* [Reynolds, 1983]

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

This creates a need for a certain degree of *type inference*.

Because type systems are *compositional*, a type inference problem can often be expressed as a *constraint solving* problem, where constraints are made up of predicates about types, conjunction, and existential quantification.

Type-preserving compilation

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

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

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

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

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

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

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

A sound type system with decidable type-checking (and possibly decidable type inference) must be *conservative*.

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."*

Outline of the course

This part of the course is structured in eight lectures:

1. Simple types / type soundness
2. Simple types / type inference
3. Polymorphism / system F and ML
4. Polymorphism / type soundness
5. Polymorphism / type inference
6. Recursive types, and more
7. Existential types
8. Type-preserving closure conversion

Why λ-calculus?

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

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

λ-terms, also known as *terms* and *expressions*, are given by:

$$t ::= x \mid \lambda x.t \mid t\ t \mid \ldots$$

where x denotes a variable.

Types are given by

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

where X denotes a type variable.

More term- and type-level constructs are introduced later on.

We use a *small-step operational* semantics.

We choose a *call-by-value* variant. When explaining *references*, *exceptions*, or other forms of side effects, this choice matters. Otherwise, most of the type-theoretic machinery applies to call-by-name or call-by-need just as well.

In the pure λ-calculus, the *values* are the functions:

$$v ::= \lambda x.t \mid \ldots$$

The *reduction relation* $t \longrightarrow t$ is inductively defined:

$$\beta_v \quad \frac{}{(\lambda x.t)\ v \longrightarrow [x \mapsto v]t} \qquad\qquad \text{Context} \quad \frac{t \longrightarrow t'}{E[t] \longrightarrow E[t']}$$

Evaluation contexts are defined as follows:

$$E ::= [\,]\ t \mid v\ [\,] \mid \ldots$$

Technically, the type system is a 3-place predicate, whose instances are called *judgements*. Judgements take the form:

$$\Gamma \vdash t : T$$

where a *type environment* Γ is a finite sequence of bindings of variables to types.

Judgements are defined inductively:

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

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

The following is a valid *type derivation*:

$$
\dfrac{
  \dfrac{
    \dfrac{
      \text{Var}\ \dfrac{}{\Gamma \vdash f : T_1 \longrightarrow T_2}
      \quad
      \text{Var}\ \dfrac{}{\Gamma \vdash x : T_1}
    }{\Gamma \vdash f\ x : T_2}\ \text{App}
    \qquad\qquad
    \dfrac{
      \dfrac{}{\Gamma \vdash f : T_1 \longrightarrow T_2}\ \text{Var}
      \quad
      \dfrac{}{\Gamma \vdash y : T_1}\ \text{Var}
    }{\Gamma \vdash f\ y : T_2}\ \text{App}
  }{f : T_1 \longrightarrow T_2; x, y : T_1 \vdash (f\ x, f\ y) : T_2 \times T_2}\ \text{Pair}
}{\emptyset \vdash \lambda fxy.(f\ x, f\ y) : (T_1 \longrightarrow T_2) \longrightarrow T_1 \longrightarrow T_1 \longrightarrow (T_2 \times T_2)}\ \text{Abs}^3
$$

(Γ stands for $(f : T_1 \longrightarrow T_2; x, y : T_1)$. Rule Pair is introduced later on.)

(*This derivation is valid for any choice of T_1 and T_2. Conversely, every derivation for this term must have this shape, for some T_1 and T_2.*)

A derived construct: let

The construct "let $x = t_1$ in t_2" can be viewed as syntactic sugar for the β-redex "$(\lambda x.t_2)\ t_1$".

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

$$\dfrac{\textsf{Abs}\ \dfrac{\Gamma; x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x.t_2 : T_1 \rightarrow T_2} \qquad \Gamma \vdash t_1 : T_1}{\Gamma \vdash (\lambda x.t_2)\ t_1 : T_2}\ \textsf{App}$$

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

$$\textsf{LetMono}\quad \dfrac{\Gamma \vdash t_1 : T_1 \qquad \Gamma; x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}$$

Contents

What is a formal statement of Milner's slogan?

> *"Well-typed expressions do not go wrong."*

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

By definition, a closed, irreducible term is either a value or *stuck*.

A closed term must *converge* to a value, *diverge*, or *go wrong* by reducing to a stuck term.

Milner's slogan now has formal meaning:

Theorem (Type Soundness)

Well-typed expressions do not go wrong.

Proof.

By Subject Reduction and Progress. ∎

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.*

This syntactic proof method is due to Wright and Felleisen [1994].

Subject reduction is proved by *structural induction* over the hypothesis $t \longrightarrow t'$. Thus, there is one case per reduction rule.

In the pure λ-calculus, there are just two such rules: β-reduction and reduction under an evaluation context.

$$\frac{}{(\lambda x.t)\ v \longrightarrow [x \mapsto v]t}\ \beta_v \qquad \frac{t \longrightarrow t'}{E[t] \longrightarrow E[t']}\ \text{Context}$$

In the β-reduction case, the first hypothesis is

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

and the goal is

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

How do we proceed?

We *decompose* the first hypothesis.

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

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

Where next?

To conclude, we only need a simple lemma:

Lemma (Value substitution)

$x : T_1 \vdash t : T_2$ *and* $\emptyset \vdash v : T_1$ *imply* $\emptyset \vdash [x \mapsto v]t : T_2$.

In plain words, replacing a formal parameter with a type-compatible actual argument preserves types.

How do we prove this lemma?

The lemma must be suitably generalized so it can be proven by *structural induction* over the typing derivation for t:

Lemma (Value substitution)

$x : T_1, \Gamma \vdash t : T_2$ *and* $x \notin \mathrm{dom}(\Gamma)$ *and* $\varnothing \vdash v : T_1$ *imply* $\Gamma \vdash [x \mapsto v]t : T_2$.

The proof is straightforward, and, at variables, exploits the fact that $\varnothing \vdash v : T_1$ implies $\Gamma \vdash v : T_1$ (this is known as *weakening*).

This closes the case of the β-reduction rule.

In the context case, the first hypothesis is

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

where E is an *evaluation context* ($E ::= [\,]\ t \mid v\ [\,] \mid \ldots$).

The second hypothesis is

$$t \longrightarrow t'$$

where, by induction hypothesis, this reduction preserves types.

The goal is

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

How do we proceed?

Type-checking is compositional: only the type of the sub-expression “in the hole” matters, not its exact form.

Lemma (Compositionality)

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

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

Proof.

By cases over E. $\square$

Using this lemma, the context case of the subject reduction theorem is immediate.

Progress ("A well-typed term t is either reducible or a value") is proved by *structural induction* over the term 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...

The case of variables is void, because *a variable is never well-typed* (it does not admit a type in the empty environment).

The case of λ-abstractions is immediate, because *a λ-abstraction is a value*.

In the case of applications, let us consider a well-typed term $t_1\ t_2$.

Then, by inversion of the type-checking rule for applications, there exist types T_1, T_2 such that $\emptyset \vdash t_1 : T_1 \to T_2$ and $\emptyset \vdash t_2 : T_1$. In particular, both t_1 and t_2 are well-typed.

Then, by inversion of the type-checking rule for applications, there exist types T_1, T_2 such that $\emptyset \vdash t_1 : T_1 \rightarrow T_2$ and $\emptyset \vdash t_2 : T_1$. In particular, both t_1 and t_2 are well-typed.

By the induction hypothesis, t_1 is *either reducible or a value* v_1. If it is reducible, then, *because* $[]\, t_2$ *is an evaluation context*, $t_1\, t_2$ is *reducible as well, and we are done.* Otherwise:

Then, by inversion of the type-checking rule for applications, there exist types T_1, T_2 such that $\emptyset \vdash t_1 : T_1 \rightarrow T_2$ and $\emptyset \vdash t_2 : T_1$. In particular, both t_1 and t_2 are well-typed.

By the induction hypothesis, t_1 is either reducible or a value v_1. If it is reducible, then, *because* $[]\, t_2$ *is an evaluation context*, $t_1\, t_2$ is reducible as well, and we are done. Otherwise:

By the induction hypothesis, t_2 is either reducible or a value v_2. If it is reducible, then, *because* $v_1\, []$ is an *evaluation context*, $v_1\, t_2$ is reducible as well, and we are done. Otherwise:

Then, by inversion of the type-checking rule for applications, there exist types T_1, T_2 such that $\varnothing \vdash t_1 : T_1 \rightarrow T_2$ and $\varnothing \vdash t_2 : T_1$. In particular, both t_1 and t_2 are well-typed.

By the induction hypothesis, t_1 is either reducible or a value v_1. If it is reducible, then, because $[]\, t_2$ is an evaluation context, $t_1\, t_2$ is reducible as well, and we are done. Otherwise:

By the induction hypothesis, t_2 is either reducible or a value v_2. If it is reducible, then, because $v_1\, []$ is an evaluation context, $v_1\, t_2$ is reducible as well, and we are done. Otherwise:

Because v_1 is a value of type $T_1 \rightarrow T_2$, it must be a λ-abstraction (*see next slide*), so $v_1\, v_2$ is a β-redex, and we are done. $\square$

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;*
- *...*

Proof.

By cases over v:

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

Because different kinds of values receive types with different head constructors, this classification is injective, and can be inverted. $\square$

Towards more complex type systems

In the pure λ-calculus, classification is trivial, because *every value is a λ-abstraction.* Progress would hold even in the absence of the well-typedness hypothesis, because *no term is stuck!*

As the programming language and type system are extended with new features, however, type soundness is no longer trivial.

Most type soundness proofs are shallow but large. Authors are tempted to skip the “easy” cases, but these may contain hidden traps!

Towards more complex type systems

Sometimes, the *combination* of two features is *unsound*, even though each feature, in isolation, is sound.

This will be illustrated in this course by the interaction between references and polymorphism in ML.

In fact, a few such combinations have been implemented, deployed, and used for some time before they were found to be unsound!

- call/cc + polymorphism in SML/NJ [Harper and Lillibridge, 1991]
- mutable records + existential quantification in Cyclone [Grossman, 2006]

Because the λ-calculus is a Turing-complete programming language, whether a program goes wrong is an *undecidable* property.

As a result, *any sound, decidable type system must be incomplete*, that is, must reject some valid programs.

Type systems can be *compared* against one another via encodings, so it is sometimes possible to prove that one system is more expressive than another.

However, whether a type system is "sufficiently expressive in practice" can only be assessed via *empirical* means.

Contents

The untyped calculus is modified as follows.

Values and *expressions* are *extended*:

$$
\begin{array}{lcl}
v & ::= & \ldots \mid () \\
t & ::= & \ldots \mid ()
\end{array}
$$

No new reduction rule is introduced.

The type system is modified as follows.

Types are *extended*:

$$T ::= \ldots \mid \mathsf{unit}$$

A typing rule is *introduced*:

$$\text{Unit}\quad \frac{}{\Gamma \vdash () : \mathsf{unit}}$$

The untyped calculus is modified as follows.

Values, expressions, evaluation contexts are extended:

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

A new reduction rule is introduced:

$$\mathsf{proj}_i\ (v_1, v_2) \longrightarrow v_i$$

The type system is modified as follows.

Types are extended:

$$T ::= \ldots \mid T \times T$$

Two new typing rules are introduced:

$$\textsc{Pair}\ \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2} \qquad\qquad \textsc{Proj}\ \frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash \mathsf{proj}_i\ t : T_i}$$

The untyped calculus is modified as follows.

Values, expressions, evaluation contexts are extended:

$$\begin{array}{rcl} v & ::= & \ldots \mid \mathsf{inj}_i\ v \\ t & ::= & \ldots \mid \mathsf{inj}_i\ t \mid \mathit{case}\ t\ \mathsf{of}\ v\ [\!]\ v \\ E & ::= & \ldots \mid \mathsf{inj}_i\ [] \mid \mathit{case}\ []\ \mathsf{of}\ v\ [\!]\ v \end{array}$$

A new reduction rule is introduced:

$$\mathsf{case}\ \mathsf{inj}_i\ v\ \mathsf{of}\ v_1\ [\!]\ v_2 \longrightarrow v_i\ v$$

The type system is modified as follows.

Types are extended:

$$T ::= \ldots \mid T + T$$

Two new typing rules are introduced:

Inj

$$\frac{\Gamma \vdash t : T_i}{\Gamma \vdash \mathsf{inj}_i\ t : T_1 + T_2}$$

Case

$$\frac{\Gamma \vdash t : T_1 + T_2 \qquad \Gamma \vdash v_1 : T_1 \longrightarrow T \qquad \Gamma \vdash v_2 : T_2 \longrightarrow T}{\Gamma \vdash \mathit{case}\ t\ \mathit{of}\ v_1 \ [] \ v_2 : T}$$

The untyped calculus is modified as follows.

Values and expressions are extended:

$$\begin{array}{rcl} v & ::= & \ldots \mid \mu f.\lambda x.t \\ t & ::= & \ldots \mid \mu f.\lambda x.t \end{array}$$

A new reduction rule is introduced:

$$(\mu f.\lambda x.t)\ v \longrightarrow [f \mapsto \mu f.\lambda x.t][x \mapsto v]t$$

The type system is modified as follows.

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

A new typing rule is introduced:

$$\text{FixAbs}\quad \frac{\Gamma; f : T_1 \rightarrow T_2 \vdash \lambda x.t : T_1 \rightarrow T_2}{\Gamma \vdash \mu f.\lambda x.t : T_1 \rightarrow T_2}$$

In the premise, the type $T_1 \rightarrow T_2$ serves both as an assumption and a goal. This is a typical feature of recursive definitions.

The construct "let rec $f\ x = t_1$ in t_2" can be viewed as syntactic sugar for "let $f = \mu f.\lambda x.t_1$ in t_2".

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

$$\text{LetMono}\ \dfrac{\text{FixAbs}\ \dfrac{\Gamma; f : T_1 \rightarrow T_1'; x : T_1 \vdash t_1 : T_1'}{\Gamma \vdash \mu f.\lambda x.t_1 : T_1 \rightarrow T_1'} \qquad \Gamma; f : T_1 \rightarrow T_1' \vdash t_2 : T_2}{\Gamma \vdash \text{let } f = \mu f.\lambda x.t_1 \text{ in } t_2 : T_2}$$

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

$$\text{LetRecMono}\ \dfrac{\Gamma; f : T_1 \rightarrow T_1'; x : T_1 \vdash t_1 : T_1' \qquad \Gamma; f : T_1 \rightarrow T_1' \vdash t_2 : T_2}{\Gamma \vdash \text{let rec } f\ x = t_1 \text{ in } t_2 : T_2}$$

In the ML vocabulary, a *reference cell*, or reference, is a dynamically allocated block of memory, which holds a value, and whose contents can change over time.

A reference can be allocated and initialized (*ref*), written (*:=*), and read (*!*).

Expressions and evaluation contexts are extended:

$$\begin{array}{rcl} t & ::= & \ldots \mid \mathsf{ref}\ t \mid t := t \mid !t \\ E & ::= & \ldots \mid \mathsf{ref}\ [] \mid [] := t \mid v := [] \mid ![] \end{array}$$

A reference allocation expression is not a value. Otherwise, by β_v, the program:

$$(\lambda x.(x := 1; !x))\ (\mathrm{ref}\ 3)$$

(which intuitively should yield 1) would reduce to:

$$(\mathrm{ref}\ 3) := 1; !(\mathrm{ref}\ 3)$$

(which intuitively yields 3).

How shall we solve this problem?

(ref 3) should first reduce to a value: the address of a fresh cell.

Not just the *content* of a cell matters, but also its *address*. Writing through one copy of the address should affect a future read via another copy.

We extend the untyped calculus with *memory locations*:

$$\begin{array}{rcl} v & ::= & \ldots \mid \ell \\ t & ::= & \ldots \mid \ell \end{array}$$

A memory location is just an atom (that is, a name). The value found at a location ℓ is obtained by indirection through a *memory* (or *store*). A memory μ is a finite mapping of locations to closed values.

A *configuration* is a pair t/μ of a term and a store.

The semantics (next slide) maintains a *no-dangling-pointers* invariant: the locations that appear in t or in the image of μ are in the domain of μ.

Initially, the store is empty, and the term contains no locations, because, by convention, memory locations cannot appear in source programs. So, the invariant holds.

The operational semantics now reduces configurations.

All existing reduction rules are augmented with a store, which they *do not touch*:

$$\begin{aligned} (\lambda x.t)\, v / \mu &\longrightarrow [x \mapsto v]t / \mu \\ E[t]/\mu &\longrightarrow E[t']/\mu' \qquad \text{if } t/\mu \longrightarrow t'/\mu' \end{aligned}$$

Three new reduction rules are added:

$$\begin{aligned} \text{ref}\, v/\mu &\longrightarrow \ell/\mu[\ell \mapsto v] \quad \text{if } \ell \notin \text{dom}(\mu) \\ \ell := v/\mu &\longrightarrow ()/\mu[\ell \mapsto v] \\ !\ell/\mu &\longrightarrow \mu(\ell)/\mu \end{aligned}$$

In the last two rules, the no-dangling-pointers invariant guarantees $\ell \in \text{dom}(\mu)$.

The type system is modified as follows.

Types are extended:

$$T ::= \ldots \mid \mathsf{ref}\ T$$

Three new typing rules are introduced:

$$\textsc{Ref}\quad \frac{\Gamma \vdash t : T}{\Gamma \vdash \mathsf{ref}\ t : \mathsf{ref}\ T} \qquad \textsc{Set}\quad \frac{\Gamma \vdash t_1 : \mathsf{ref}\ T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 := t_2 : \mathsf{unit}} \qquad \textsc{Get}\quad \frac{\Gamma \vdash t : \mathsf{ref}\ T}{\Gamma \vdash !t : T}$$

Is that all we need?

The preceding slides are enough to typecheck *source terms*, but do not allow stating or proving type soundness.

Indeed, we have not yet answered these questions:

- what is the type of a memory location ℓ?
- when is a configuration t/μ well-typed?

When does a location ℓ have type ref T?

A possible answer is, "when it points to some value of type T". This would be formalized by a typing rule of the form:

$$\frac{\mu, \emptyset \vdash \mu(\ell) : T}{\mu, \Gamma \vdash \ell : \mathsf{ref}\ T}$$

Comments?

When does a location ℓ have type ref T?

A possible answer is, "when it points to some value of type T". This would be formalized by a typing rule of the form:

$$\frac{\mu, \emptyset \vdash \mu(\ell) : T}{\mu, \Gamma \vdash \ell : \mathsf{ref}\ T}$$

Comments?

- typing judgements would have the form $\mu, \Gamma \vdash t : T$.

When does a location ℓ have type $\mathsf{ref}\ T$?

A possible answer is, “when it points to some value of type T”. This would be formalized by a typing rule of the form:

$$\frac{\mu, \emptyset \vdash \mu(\ell) : T}{\mu, \Gamma \vdash \ell : \mathsf{ref}\ T}$$

Comments?

- typing judgements would have the form $\mu, \Gamma \vdash t : T$.
- typing judgements would no longer be *inductively* defined (or else, every cyclic structure would be ill-typed). Instead, *co-induction* would be required.

When does a location ℓ have type ref T?

A possible answer is, "when it points to some value of type T". This would be formalized by a typing rule of the form:

$$\frac{\mu, \emptyset \vdash \mu(\ell) : T}{\mu, \Gamma \vdash \ell : \text{ref } T}$$

Comments?

- typing judgements would have the form $\mu, \Gamma \vdash t : T$.
- typing judgements would no longer be *inductively* defined (or else, every cyclic structure would be ill-typed). Instead, *co-induction* would be required.
- if the value $\mu(\ell)$ happens to admit two distinct types T_1 and T_2, then ℓ admits types ref T_1 and ref T_2. So, one can write at type T_1 and read at type T_2: this rule is *unsound!*

A simpler, and sound, approach is to fix the type of a memory location when it is first allocated. To do so, we use a *store typing* M, a finite mapping of locations to types.

So, when does a location ℓ have type ref T? "When M says so."

$$\text{Loc} \quad \frac{}{M, \Gamma \vdash \ell : \mathsf{ref}\ M(\ell)}$$

Comments:

- typing judgements now have the form $M, \Gamma \vdash t : T$.

How do we know that the store typing predicts appropriate types?

This is required by the typing rules for stores and configurations:

$$\text{Store}\quad \frac{\forall \ell \in \mathrm{dom}(\mu), \quad M, \emptyset \vdash \mu(\ell) : M(\ell)}{\vdash \mu : M} \qquad\qquad \text{Config}\quad \frac{M, \emptyset \vdash t : T \qquad \vdash \mu : M}{\vdash t / \mu : T}$$

Comments:

- This is an *inductive* definition. The store typing M serves both as an assumption (Loc) and a goal (Store). Cyclic stores are not a problem.
- The store typing exists neither at runtime nor at type-checking time. It is used only in the definition of a “well-typed configuration” and in the type soundness proof.

The type soundness statements are slightly modified:

Theorem (Subject reduction)

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

Theorem (Progress)

If t/μ *is a well-typed, irreducible configuration, then* t *is a value.*

By definition (see Config), subject reduction can also be written:

Theorem (Subject reduction, detailed)

Assume $M, \emptyset \vdash t : T$ and $\vdash \mu : M$ and $t/\mu \longrightarrow t'/\mu'$. Then, there exists M' such that $M', \emptyset \vdash t' : T$ and $\vdash \mu' : M'$.

This statement is correct, but *too weak* — its proof by induction will fail in one case. (Which?)

Let us look at the case of reduction under a context.

The hypotheses are:

$$M, \emptyset \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

Let us look at the case of reduction under a context.

The hypotheses are:

$$M, \emptyset \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t / \mu \longrightarrow t' / \mu'$$

By compositionality (?), there exists T' such that:

$$M, \emptyset \vdash t : T' \quad \text{and} \quad \forall t', \quad (M, \emptyset \vdash t' : T') \Rightarrow (M, \emptyset \vdash E[t'] : T)$$

Let us look at the case of reduction under a context.

The hypotheses are:

$$M, \emptyset \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

By compositionality (?), there exists T' such that:

$$M, \emptyset \vdash t : T' \quad \text{and} \quad \forall t', \quad (M, \emptyset \vdash t' : T') \Rightarrow (M, \emptyset \vdash E[t'] : T)$$

By the induction hypothesis, there exists M' such that:

$$M', \emptyset \vdash t' : T' \quad \text{and} \quad \vdash \mu' : M'$$

Let us look at the case of reduction under a context.

The hypotheses are:

$$M, \emptyset \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

By compositionality (?), there exists T' such that:

$$M, \emptyset \vdash t : T' \quad \text{and} \quad \forall t', \quad (M, \emptyset \vdash t' : T') \Rightarrow (M, \emptyset \vdash E[t'] : T)$$

By the induction hypothesis, there exists M' such that:

$$M', \emptyset \vdash t' : T' \quad \text{and} \quad \vdash \mu' : M'$$

Here, *we are stuck*. The context E is well-typed under M, but the term t' is well-typed under M', so we cannot combine them. How could we fix this?

We are missing a key property: *the store typing grows with time*. That is, although new memory locations can be allocated, *the type of an existing location does not change*.

This is formalized by strengthening the subject reduction statement:

Theorem (Subject reduction, strengthened)

Assume $M, \emptyset \vdash t : T$ and $\vdash \mu : M$ and $t / \mu \longrightarrow t' / \mu'$. Then, there exists M' such that $M', \emptyset \vdash t' : T$ and $\vdash \mu' : M'$ and $M \subseteq M'$.

At each reduction step, the new store typing M' extends the previous store typing M.

Growing the store typing preserves well-typedness:

Lemma (Stability under memory allocation)

$M, \Gamma \vdash t : T$ *and* $M \subseteq M'$ *imply* $M', \Gamma \vdash t : T$.

Stability under memory allocation allows establishing a strengthened version of compositionality:

Lemma (Compositionality)

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

- $M, \emptyset \vdash t : T'$,
- *for every M' such that $M \subseteq M'$,*
  for every t',
  $M', \emptyset \vdash t' : T'$ *implies* $M', \emptyset \vdash E[t'] : T$.

Let us now look again at the case of reduction under a context.

The hypotheses are:

$$M, \emptyset \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

Let us now look again at the case of reduction under a context.

The hypotheses are:

$$M, \emptyset \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

By compositionality, there exists T' such that:

$$\begin{array}{l} M, \emptyset \vdash t : T' \\ \forall M', \forall t', \quad (M \subseteq M') \Rightarrow (M', \emptyset \vdash t' : T') \Rightarrow (M', \emptyset \vdash E[t'] : T') \end{array}$$

Let us now look again at the case of reduction under a context.

The hypotheses are:

$$M, \varnothing \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

By compositionality, there exists T' such that:

$$\begin{array}{l} M, \varnothing \vdash t : T' \\ \forall M', \forall t', \quad (M \subseteq M') \Rightarrow (M', \varnothing \vdash t' : T') \Rightarrow (M', \varnothing \vdash E[t'] : T') \end{array}$$

By the induction hypothesis, there exists M' such that:

$$M', \varnothing \vdash t' : T' \quad \text{and} \quad \vdash \mu' : M' \quad \text{and} \quad M \subseteq M'$$

The goal follows immediately.

Exercise (Recommended)

Prove subject reduction and progress for simply-typed λ-calculus equipped with unit, pairs, sums, recursive functions, and references.

Haskell adopts a different route and chooses to distinguish effectful computations [Peyton Jones and Wadler, 1993, Peyton Jones, 2009].

$$
\begin{aligned}
\mathsf{return} &: X \longrightarrow \mathsf{IO}\ X \\
\mathsf{bind} &: \mathsf{IO}\ X \longrightarrow (X \longrightarrow \mathsf{IO}\ Y) \longrightarrow \mathsf{IO}\ Y \\[1ex]
\mathsf{main} &: \mathsf{IO}\ () \\[1ex]
\mathsf{newIORef} &: X \longrightarrow \mathsf{IO}\ (\mathsf{IORef}\ X) \\
\mathsf{readIORef} &: \mathsf{IORef}\ X \longrightarrow \mathsf{IO}\ X \\
\mathsf{writeIORef} &: \mathsf{IORef}\ X \longrightarrow X \longrightarrow \mathsf{IO}\ ()
\end{aligned}
$$

Haskell offers many monads other than IO. In particular, the ST monad offers references whose lifetime is statically controlled.

On memory deallocation

In ML, memory deallocation is implicit. It must be performed by the runtime system, possibly with the cooperation of the compiler.

The most common technique is *garbage collection*. A more ambitious technique, implemented in the ML Kit, is compile-time *region analysis* [Tofte et al., 2004].

References in ML are easy to type-check, thanks in large part to the *no-dangling-pointers* property of the semantics.

Making memory deallocation an explicit operation, while preserving type soundness, is possible, but difficult. This requires reasoning about *aliasing* and *ownership*. See Charguéraud and Pottier's recent paper [2008] for citations.

For a textbook introduction to λ-calculus and simple types, see Pierce [2002].

For more details about syntactic type soundness proofs, see Wright and Felleisen [1994].

Contents

Logical versus algorithmic properties

We have viewed a type system as a 3-place *predicate* over a type environment, a term, and a type.

So far, we have been concerned with *logical* properties of the type system, namely subject reduction and progress.

However, one should also study its *algorithmic* properties: is it decidable whether a term is well-typed? If not, can the problem be made decidable by requesting programmers to *annotate* programs with explicit type information?

The typing judgement is *inductively defined*, so that, in order to prove that a particular instance holds, one exhibits a *type derivation*.

In the case of simply-typed λ-calculus, a type derivation is essentially a version of the program where every node is annotated with a type.

Checking that a type derivation is correct is easy: it basically amounts to checking equalities between types.

However, type derivations are so verbose as to be intractable by humans! Requiring every node to be type-annotated is not practical.

A more practical, and quite common, approach consists in requesting just enough annotations to allow types to be reconstructed in a *bottom-up* manner.

In other words, one seeks an *algorithmic reading* of the typing rules, where, in a judgement $\Gamma \vdash t : T$, the parameters Γ and t are *inputs*, while the parameter T is an *output*.

In the pure, simply-typed λ-calculus, this is quite easy, provided *every λ-bound variable carries a type*, that is, provided λ-abstractions take the form $\lambda x : T.t$ (sometimes known as “Church’s style”).

This is the traditional approach of Pascal, C, C++, Java, ...: formal procedure parameters, as well as local variables, are assigned explicit types. The types of expressions are synthesized bottom-up.

Unfortunately, bottom-up type checking does not work for some of the typing rules that we have presented (FixAbs, Inj). Annotations would be required there as well.

This seems cumbersome. Perhaps one would prefer to program "Curry style", without annotations.

For simply-typed λ-calculus, it turns out that this is possible: *whether a term is well-typed is decidable*, even when no type annotations are provided!

This algorithm, due to Hindley [1969], is known as *type inference*.

The idea behind Hindley's type inference algorithm is simple.

Because simply-typed λ-calculus is a *syntax-directed* type system, an unannotated term determines an isomorphic *candidate type derivation*, where all types are unknown: they are distinct *type variables*.

For a candidate type derivation to become an actual, valid type derivation, every type variable must be instantiated with a type, subject to certain *equality constraints* on types.

For instance, at an application node, the type of the operator must match the domain type of the operator.

Thus, type inference for the simply-typed λ-calculus decomposes into *constraint generation* followed by *constraint solving*.

Simple types are *first-order terms.* Thus, solving a collection of equations between simple types is *first-order unification.*

First-order unification can be performed incrementally in quasi-linear time, and admits particularly simple *solved forms*.

At the interface between the constraint generation and constraint solving phases is the *constraint language*.

It is a *logic:* a *syntax*, equipped with an *interpretation* in a model.

There are two syntactic categories: *types* and *constraints*.

$$\begin{array}{rcl} T & ::= & X \mid F\,\vec{T} \\ C & ::= & \mathsf{true} \mid \mathsf{false} \mid T = T \mid C \wedge C \mid \exists X.C \end{array}$$

A type is either a *type variable* X or an arity-consistent application of a *type constructor* F.

(The type constructors are unit, $\times$, $+$, $\rightarrow$, *etc.*)

An atomic constraint is truth, falsity, or an *equation* between types. Compound constraints are built on top of atomic constraints via *conjunction* and *existential quantification* over type variables.

Constraints are interpreted in the Herbrand universe, that is, in the set of *ground types:*

$$\mathbf{t} \quad ::= \quad F\,\vec{\mathbf{t}}$$

Ground types contain no variables. The base case in this definition is when F has arity zero.

A *ground assignment* ϕ is a total mapping of type variables to ground types.

A ground assignment determines a total mapping of types to ground types.

The interpretation of constraints takes the form of a judgement, $\phi \vdash C$, pronounced: *ϕ satisfies C*, or *ϕ is a solution of C*.

This judgement is inductively defined:

$$\phi \vdash \text{true} \qquad \frac{\phi T_1 = \phi T_2}{\phi \vdash T_1 = T_2} \qquad \frac{\phi \vdash C_1 \qquad \phi \vdash C_2}{\phi \vdash C_1 \wedge C_2} \qquad \frac{\phi[X \mapsto \mathbf{t}] \vdash C}{\phi \vdash \exists X.C}$$

A constraint C is *satisfiable* if and only if there exists a ground assignment ϕ that satisfies C.

I write $C_1 \equiv C_2$ when C_1 and C_2 have the same solutions.

The problem: "given a constraint C, is C satisfiable?" is *first-order unification*.

Type inference is reduced to constraint solving by defining a mapping of *candidate judgements* to constraints.

$$
\begin{aligned}
[\![\Gamma \vdash x : T]\!] &= \Gamma(x) = T \\
[\![\Gamma \vdash \lambda x.t : T]\!] &= \exists X_1 X_2.([\![\Gamma; x : X_1 \vdash t : X_2]\!] \wedge X_1 \rightarrow X_2 = T) \\
&\quad \text{if } X_1, X_2 \# \Gamma, t, T \\
[\![\Gamma \vdash t_1\ t_2 : T]\!] &= \exists X.([\![\Gamma \vdash t_1 : X \rightarrow T]\!] \wedge [\![\Gamma \vdash t_2 : X]\!]) \\
&\quad \text{if } X \# \Gamma, t_1, t_2, T
\end{aligned}
$$

Thanks to the use of existential quantification, the names that occur free in $[\![\Gamma \vdash t : T]\!]$ are a subset of those that occur free in Γ or T.

This allows the freshness side-conditions to remain *local* — there is no need to informally require “globally fresh” type variables.

Let us perform type inference for the closed term

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

The problem is to *construct* and *solve* the constraint

$$[\![\emptyset \vdash \lambda fxy.(f\ x, f\ y) : X_0]\!]$$

It is possible (and, for a human, easier) to mix these tasks. A machine, however, could generate and solve in two successive phases.

$$\llbracket \emptyset \vdash \lambda fxy.(f\ x, f\ y) : X_0 \rrbracket$$

$$= \exists X_1 X_2. \left(\begin{array}{l} \llbracket f : X_1 \vdash \lambda xy.\ldots : X_2 \rrbracket \\ X_1 \rightarrow X_2 = X_0 \end{array} \right)$$

$$= \exists X_1 X_2. \left(\begin{array}{l} \exists X_3 X_4. \left(\begin{array}{l} \llbracket f : X_1; x : X_3 \vdash \lambda y.\ldots : X_4 \rrbracket \\ X_3 \rightarrow X_4 = X_2 \end{array} \right) \\ X_1 \rightarrow X_2 = X_0 \end{array} \right)$$

$$= \exists X_1 X_2. \left(\begin{array}{l} \exists X_3 X_4. \left(\begin{array}{l} \exists X_5 X_6. \left(\begin{array}{l} \llbracket f : X_1; x : X_3; y : X_5 \vdash (f\ x, f\ y) : X_6 \rrbracket \\ X_5 \rightarrow X_6 = X_4 \end{array} \right) \\ X_3 \rightarrow X_4 = X_2 \end{array} \right) \\ X_1 \rightarrow X_2 = X_0 \end{array} \right)$$

We have performed constraint generation for the 3 λ-abstractions.

$$\exists X_1 X_2.\left(\begin{array}{l} \exists X_3 X_4.\left(\begin{array}{l} \exists X_5 X_6.\left(\begin{array}{l} [\![f : X_1; x : X_3; y : X_5 \vdash (f\ x, f\ y) : X_6]\!] \\ X_5 \longrightarrow X_6 = X_4 \end{array}\right) \\ X_3 \longrightarrow X_4 = X_2 \end{array}\right) \\ X_1 \longrightarrow X_2 = X_0 \end{array}\right)$$

$$\equiv \quad \exists X_1 X_2 X_3 X_4 X_5 X_6.\left(\begin{array}{l} [\![f : X_1; x : X_3; y : X_5 \vdash (f\ x, f\ y) : X_6]\!] \\ X_5 \longrightarrow X_6 = X_4 \\ X_3 \longrightarrow X_4 = X_2 \\ X_1 \longrightarrow X_2 = X_0 \end{array}\right)$$

We have hoisted up several existential quantifiers:

$$(\exists X.C_1) \wedge C_2 \equiv \exists X.(C_1 \wedge C_2) \qquad \text{if } X \# C_2$$

$$\exists X_1 X_2 X_3 X_4 X_5 X_6. \left(\begin{array}{l} [\![f : X_1; x : X_3; y : X_5 \vdash (f\ x, f\ y) : X_6]\!] \\ X_5 \rightarrow X_6 = X_4 \\ X_3 \rightarrow X_4 = X_2 \\ X_1 \rightarrow X_2 = X_0 \end{array} \right)$$

$$\equiv \quad \exists X_1 X_2 X_3 X_5 X_6. \left(\begin{array}{l} [\![f : X_1; x : X_3; y : X_5 \vdash (f\ x, f\ y) : X_6]\!] \\ X_3 \rightarrow X_5 \rightarrow X_6 = X_2 \\ X_1 \rightarrow X_2 = X_0 \end{array} \right)$$

We have eliminated a type variable (X_4) with a defining equation:

$$\exists X.(C \wedge X = T) \equiv [X \mapsto T]C \qquad \text{if } X \# T$$

$$\exists X_1 X_2 X_3 X_5 X_6.\left(\begin{array}{l} [\![f : X_1; x : X_3; y : X_5 \vdash (f\ x, f\ y) : X_6]\!] \\ X_3 \rightarrow X_5 \rightarrow X_6 = X_2 \\ X_1 \rightarrow X_2 = X_0 \end{array}\right)$$

$$\equiv \quad \exists X_1 X_3 X_5 X_6.\left(\begin{array}{l} [\![f : X_1; x : X_3; y : X_5 \vdash (f\ x, f\ y) : X_6]\!] \\ X_1 \rightarrow X_3 \rightarrow X_5 \rightarrow X_6 = X_0 \end{array}\right)$$

We have again eliminated a type variable (X_2) with a defining equation.

In the following, let Γ stand for $(f : X_1; x : X_3; y : X_5)$.

$$\exists X_1 X_3 X_5 X_6. \left(\begin{array}{l} [\![\Gamma \vdash (f\ x, f\ y) : X_6]\!] \\ X_1 \to X_3 \to X_5 \to X_6 = X_0 \end{array} \right)$$

$$\equiv \quad \exists X_1 X_3 X_5 X_6 X_7 X_8. \left(\begin{array}{l} [\![\Gamma \vdash f\ x : X_7]\!] \\ [\![\Gamma \vdash f\ y : X_8]\!] \\ X_7 \times X_8 = X_6 \\ X_1 \to X_3 \to X_5 \to X_6 = X_0 \end{array} \right)$$

$$\equiv \quad \exists X_1 X_3 X_5 X_7 X_8. \left(\begin{array}{l} [\![\Gamma \vdash f\ x : X_7]\!] \\ [\![\Gamma \vdash f\ y : X_8]\!] \\ X_1 \to X_3 \to X_5 \to X_7 \times X_8 = X_0 \end{array} \right)$$

We have performed constraint generation for the pair, hoisted the resulting existential quantifiers, and eliminated a type variable (X_6).

Let us now focus on the left-hand application...

$$
\begin{array}{rl}
& [\![\Gamma \vdash f\ x : X_7]\!] \\
= & \exists X_9. \left(\begin{array}{l} [\![\Gamma \vdash f : X_9 \rightarrow X_7]\!] \\ [\![\Gamma \vdash x : X_9]\!] \end{array} \right) \\
\\
= & \exists X_9. \left(\begin{array}{l} X_1 = X_9 \rightarrow X_7 \\ X_3 = X_9 \end{array} \right) \\
\\
\equiv & X_1 = X_3 \rightarrow X_7
\end{array}
$$

We have performed constraint generation for the variables f and x, and eliminated a type variable (X_9).

Recall that Γ stands for $(f : X_1; x : X_3; y : X_5)$.

Now, back to the big picture...

$$\exists X_1 X_3 X_5 X_7 X_8. \left(\begin{array}{l} [\![\Gamma \vdash f\ x : X_7]\!] \\ [\![\Gamma \vdash f\ y : X_8]\!] \\ X_1 \rightarrow X_3 \rightarrow X_5 \rightarrow X_7 \times X_8 = X_0 \end{array} \right)$$

$$\equiv \quad \exists X_1 X_3 X_5 X_7 X_8. \left(\begin{array}{l} X_1 = X_3 \rightarrow X_7 \\ [\![\Gamma \vdash f\ y : X_8]\!] \\ X_1 \rightarrow X_3 \rightarrow X_5 \rightarrow X_7 \times X_8 = X_0 \end{array} \right)$$

$$\equiv \quad \exists X_1 X_3 X_5 X_7 X_8. \left(\begin{array}{l} X_1 = X_3 \rightarrow X_7 \\ X_1 = X_5 \rightarrow X_8 \\ X_1 \rightarrow X_3 \rightarrow X_5 \rightarrow X_7 \times X_8 = X_0 \end{array} \right)$$

We have applied the previous simplification under a context:

$$C_1 \equiv C_2 \Rightarrow \mathcal{C}[C_1] \equiv \mathcal{C}[C_2]$$

We have simplified the right-hand application analogously.

$$\exists X_1 X_3 X_5 X_7 X_8.\left(\begin{array}{l} X_1 = X_3 \rightarrow X_7 \\ X_1 = X_5 \rightarrow X_8 \\ X_1 \rightarrow X_3 \rightarrow X_5 \rightarrow X_7 \times X_8 = X_0 \end{array}\right)$$

$$\equiv \quad \exists X_1 X_3 X_5 X_7 X_8.\left(\begin{array}{l} X_1 = X_3 \rightarrow X_7 \\ X_3 = X_5 \\ X_7 = X_8 \\ X_1 \rightarrow X_3 \rightarrow X_5 \rightarrow X_7 \times X_8 = X_0 \end{array}\right)$$

$$\equiv \quad \exists X_3 X_7.\left((X_3 \rightarrow X_7) \rightarrow X_3 \rightarrow X_3 \rightarrow X_7 \times X_7 = X_0 \right)$$

We have applied transitivity at X_1, structural decomposition, and eliminated three type variables (X_1, X_5, X_8).

We have now reached a *solved form*.

We have checked the following equivalence:

$$\begin{array}{ll} & [\![\varnothing \vdash \lambda fxy.(f\ x, f\ y) : X_0]\!] \\ \equiv & \exists X_3 X_7.\big(\ (X_3 \rightarrow X_7) \rightarrow X_3 \rightarrow X_3 \rightarrow X_7 \times X_7 = X_0\ \big) \end{array}$$

The ground types of $\lambda fxy.(f\ x, f\ y)$ are all ground types of the form $(\mathbf{t}_3 \rightarrow \mathbf{t}_7) \rightarrow \mathbf{t}_3 \rightarrow \mathbf{t}_3 \rightarrow \mathbf{t}_7 \times \mathbf{t}_7$.

$(X_3 \rightarrow X_7) \rightarrow X_3 \rightarrow X_3 \rightarrow X_7 \times X_7$ is a *principal type* for $\lambda fxy.(f\ x, f\ y)$.

Objective Caml implements a form of this type inference algorithm:

```
# fun f x y -> (f x, f y);;
- : ('a -> 'b) -> 'a -> 'a -> 'b * 'b = <fun>
```

This technique is used also by Standard ML and Haskell.

An example

In the simply-typed λ-calculus, type inference works just as well for *open* terms. Consider, for instance:

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

This term has a free variable, namely f.

The type inference problem is to *construct* and *solve* the constraint

$$[\![f : X_1 \vdash \lambda xy.(f\ x, f\ y) : X_2]\!]$$

We have already done so... with only a slight difference: X_1 and X_2 are now free, so they cannot be eliminated.

One can check the following equivalence:

$$
\begin{aligned}
& [\![f : X_1 \vdash \lambda xy.(f\ x, f\ y) : X_2]\!] \\
\equiv\quad & \exists X_3 X_7. \left(\begin{array}{l} X_3 \longrightarrow X_7 = X_1 \\ X_3 \longrightarrow X_3 \longrightarrow X_7 \times X_7 = X_2 \end{array} \right)
\end{aligned}
$$

In other words, the ground *typings* of $\lambda xy.(f\ x, f\ y)$ are all ground typings of the form:

$$
((f : \mathbf{t}_3 \longrightarrow \mathbf{t}_7), \mathbf{t}_3 \longrightarrow \mathbf{t}_3 \longrightarrow \mathbf{t}_7 \times \mathbf{t}_7)
$$

A typing is a pair of an environment and a type.

Definition

(Γ, T) is a *typing* of t if and only if $\mathrm{dom}(\Gamma) = \mathrm{fv}(t)$ and the judgement $\Gamma \vdash t : T$ is valid.

The type inference problem is to determine whether a term t admits a typing, and, if possible, to exhibit a description of the set of all of its typings.

Up to a change of universes, the problem reduces to finding the *ground typings* of a term. (For every type variable, introduce a nullary type constructor. Then, ground typings in the extended universe are in one-to-one correspondence with typings in the original universe.)

Theorem (Soundness and completeness)

$\phi \vdash [\![\Gamma \vdash t : T]\!]$ if and only if $\phi\Gamma \vdash t : \phi T$.

Proof.

By structural induction over t. (Recommended exercise.) $\square$

In other words, assuming $\mathrm{dom}(\Gamma) = \mathrm{fv}(t)$, ϕ satisfies the constraint $[\![\Gamma \vdash t : T]\!]$ if and only if $(\phi\Gamma, \phi T)$ is a (ground) typing of t.

Corollary

Let $\mathrm{fv}(t) = \{x_1, \ldots, x_n\}$*, where* $n \geq 0$*. Let* $X_0, \ldots, X_n$ *be pairwise distinct type variables. Then, the ground typings of* t *are described by*

$$((x_i : \phi X_i)_{1 \geq i \geq n}, \phi X_0)$$

where ϕ *ranges over all solutions of* $[\![(x_i : X_i)_{1 \geq i \geq n} \vdash t : X_0]\!]$.

Corollary

Let $\mathrm{fv}(t) = \emptyset$*. Then,* t *is well-typed if and only if* $\exists X.[\![\emptyset \vdash t : X]\!] \equiv \mathrm{true}$.

A constraint solving algorithm is typically presented as a (nondeterministic) system of *constraint rewriting rules*.

The system must enjoy the following properties:

- reduction is meaning-preserving: $C_1 \longrightarrow C_2$ implies $C_1 \equiv C_2$;
- reduction is terminating;
- *every* normal form is either "false" (literally) *or* satisfiable.

The normal forms are called *solved forms*.

Following Pottier and Rémy [2005, §10.6], I *extend* the syntax of constraints and replace ordinary binary equations with *multi-equations:*

$$U ::= \mathrm{true} \mid \mathrm{false} \mid \epsilon \mid U \wedge U \mid \exists \bar{X}.U$$

A multi-equation ϵ is a multi-set of types. Its interpretation is:

$$\frac{\forall T \in \epsilon, \quad \phi T = \mathbf{t}}{\phi \vdash \epsilon}$$

That is, ϕ satisfies ϵ if and only if ϕ maps all members of ϵ to a single ground type.

First-order unification as constraint solving

$$
\begin{array}{rcll}
(\exists \bar{X}.U_1) \wedge U_2 & \longrightarrow & \exists \bar{X}.(U_1 \wedge U_2) & \text{(extrusion)} \\
 & & \text{if } \bar{X} \# U_2 & \\
X = \epsilon \wedge X = \epsilon' & \longrightarrow & X = \epsilon = \epsilon' & \text{(fusion)} \\
F\,\vec{X} = F\,\vec{T} = \epsilon & \longrightarrow & \vec{X} = \vec{T} \wedge F\,\vec{X} = \epsilon & \text{(decomposition)} \\
F\,T_1 \ldots T_i \ldots T_n = \epsilon & \longrightarrow & \exists X.(X = T_i \wedge F\,T_1 \ldots X \ldots T_n = \epsilon) & \text{(naming)} \\
 & & \text{if } T_i \text{ is not a variable} \wedge X \# T_1, \ldots, T_n, \epsilon & \\
F\,\vec{T} = F'\,\vec{T}' = \epsilon & \longrightarrow & \text{false} & \text{(clash)} \\
 & & \text{if } F \neq F' & \\
U & \longrightarrow & \text{false} & \text{(occurs check)} \\
 & & \text{if } U \text{ is cyclic} & \\
\mathcal{U}[\text{false}] & \longrightarrow & \text{false} & \text{(error propag.)}
\end{array}
$$

X *dominates* Y (with respect to U) iff U contains a multi-equation of the form $F\,T_1 \ldots Y \ldots T_n = X = \ldots$

U is *cyclic* iff its domination relation is cyclic.

A cyclic constraint is unsatisfiable: indeed, if ϕ satisfies U and if X is a member of a cycle, then the ground type ϕX must be a strict subterm of itself, a contradiction.

A solved form is either false or $\exists \bar{X}.U$, where U is a conjunction of multi-equations, every multi-equation contains at most one non-variable term, no two multi-equations share a variable, and the domination relation is acyclic.

Every solved form of the second variety is satisfiable — indeed, a solution is easily constructed by well-founded recursion over the domination relation.

Viewing a unification algorithm as a system of rewriting rules makes it easy to explain and reason about.

In practice, following Huet [1976], first-order unification is implemented on top of an efficient *union-find* data structure [Tarjan, 1975]. Its time complexity is quasi-linear.

Thanks to type inference, *conciseness* and *static safety* are not incompatible.

Furthermore, an inferred type is sometimes *more general* than a programmer-intended type. Type inference helps reveal unexpected generality.

(Most titles are clickable links to online versions.)

Charguéraud, A. and Pottier, F. 2008.
Functional translation of a calculus of capabilities.
In *ACM International Conference on Functional Programming (ICFP)*. 213–224.

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

Grossman, D. 2006.
Quantified types in an imperative language.
ACM Transactions on Programming Languages and Systems 28, 3 (May), 429–475.

Bibliography]Bibliography

Harper, B. and Lillibridge, M. 1991.
ML with callcc is unsound.
Message to the TYPES mailing list.

Hindley, J. R. 1969.
The principal type-scheme of an object in combinatory logic.
Transactions of the American Mathematical Society 146, 29–60.

Huet, G. 1976.
Résolution d'équations dans des langages d'ordre 1, 2, . . ., ω.
Ph.D. thesis, Université Paris 7.

Hughes, J. 1989.
Why functional programming matters.
Computer Journal 32, 2, 98–107.

[][

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

Milner, R. 1978.
A theory of type polymorphism in programming.
Journal of Computer and System Sciences 17, 3 (Dec.), 348–375.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999.
From system F to typed assembly language.
ACM Transactions on Programming Languages and Systems 21, 3 (May), 528–569.

[][

Peyton Jones, S. 2009.
Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell.
Online lecture notes.

Peyton Jones, S. and Wadler, P. 1993.
Imperative functional programming.
In *ACM Symposium on Principles of Programming Languages (POPL)*. 71–84.

Pierce, B. C. 2002.
Types and Programming Languages.
MIT Press.

[][

Pottier, F. and Rémy, D. 2005.
The essence of ML type inference.
In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, Chapter 10, 389–489.

Reynolds, J. C. 1983.
Types, abstraction and parametric polymorphism.
In *Information Processing 83*. Elsevier Science, 513–523.

Reynolds, J. C. 1985.
Three approaches to type structure.
In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*. Lecture Notes in Computer Science, vol. 185. Springer, 97–138.

[][

Tarjan, R. E. 1975.
Efficiency of a good but not linear set union algorithm.
Journal of the ACM 22, 2 (Apr.), 215–225.

Tofte, M., Birkedal, L., Elsman, M., and Hallenberg, N. 2004.
A retrospective on region-based memory management.
Higher-Order and Symbolic Computation 17, 3 (Sept.), 245–265.

Wright, A. K. and Felleisen, M. 1994.
A syntactic approach to type soundness.
Information and Computation 115, 1 (Nov.), 38–94.