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

Didier Rémy

Plan of the course

Simply typed lambda-calculus

Metatheory of System F

ADTs, Recursive types, Existential types, GATDs

Going higher order with F^{ω}!

Logical relations

Side effects, References, Value restriction

Type reconstruction

Overloading

Metatheory of System F

Proofs

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

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

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

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

Some of the metatheory will be done in Coq, by François, Pottier,
—for your help or curiosity,

What are types?

- Types are:
  “a concise, formal description of the behavior of a program fragment.”

- Types must be *sound*:
  programs must behave as prescribed by their types.

- Hence, types must be *checked* and ill-typed programs must be rejected.

What are they useful for?

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

Type-preserving compilation

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

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

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

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

Typed or untyped?

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

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

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

Typed, Sir! with better types.

In fact, Reynolds settles the debate:

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

Today, the question is more whether

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

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

Contents

Why λ-calculus?

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

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

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

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

Simply typed λ-calculus

Why?

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

Types are:

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

Terms are:

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

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

Binders, α-conversion, and substitutions

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

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

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

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

Terms are considered equal up to renaming of bound variables:

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

Substitution:

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

Dynamic semantics

We use a *small-step operational* semantics.

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

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

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

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

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

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

Type soundness for full reduction is a stronger result.

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

Dynamic semantics

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

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

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

$$\overset{\beta_v}{(\lambda x\!:\!\tau.\, M)\ V \longrightarrow [x \mapsto V]M} \qquad\qquad \overset{\textsc{Context}}{\dfrac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}}$$

Evaluation contexts are defined as follows:

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

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

An evaluation context of arbitrary depth can be defined as:

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

Static semantics

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

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

where Γ is a typing context.

Typing context, notations

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

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

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

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

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

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

Static semantics

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

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

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

Notice that the specification is extremely simple.

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

Example

The following is a valid *typing derivation*:

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

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

Observe that:

– this is in fact, the only typing derivation (in the empty environment).

– this derivation is valid for any choice of τ and τ'
(which in our setting are part of the source term)

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

Inversion of typing rules

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

Lemma (Inversion of typing rules)

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

– *If* M *is a variable* x*, then* $x \in \mathrm{dom}(\Gamma)$ *and* $\Gamma(x) = \tau$.

– *If* M *is* $M_1\ M_2$ *then* $\Gamma \vdash M_1 : \tau_2 \rightarrow \tau$ *and* $\Gamma \vdash M_2 : \tau_2$ *for some type* τ_2.

– *If* M *is* $\lambda x{:}\tau_2.\, M_1$*, then* τ *is of the form* $\tau_2 \rightarrow \tau_1$ *and* $\Gamma, x : \tau_2 \vdash M_1 : \tau_1$.

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

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

Uniqueness of typing derivations

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

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

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

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

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

Explicitly *v.s.* implicitly typed?

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

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

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

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

Type erasure

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

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

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

Type reconstruction

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

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

Type reconstruction . . . may be partial

Annotating programs with types can lead to redundancy.

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

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

Full type reconstruction is undecidable for expressive type systems.

Some type annotations are required or type reconstruction is incomplete.

Untyped semantics

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

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

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

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

Untyped semantics

Obsviously, type erasure preserves reduction.

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

Lemma (Direct simulation)

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

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

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

Lemma (Inverse simulation)

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

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

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

Untyped semantics

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

It then has an implicitly-typed presentation.

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

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

This is the path we choose in this course.

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

Contents

Stating type soundness

What is a formal statement of the slogan

"Well-typed expressions do not go wrong"

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

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

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

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

Stating type soundness

The slogan now has a formal meaning:

Theorem (Type soundness)

Well-typed expressions do not go wrong.

Proof.

By Subject Reduction and Progress. □

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

Establishing type soundness

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

Theorem (Subject reduction)

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

Theorem (Progress)

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

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

Contents

Adding a unit

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

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

No new reduction rule is introduced.

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

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

Pairs

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

Values, expressions, evaluation contexts are extended:

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

A new reduction rule is introduced:

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

Pairs

Types are extended:

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

Two new typing rules are introduced:

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

Sums

Values, expressions, evaluation contexts are extended:

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

A new reduction rule is introduced:

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

Sums

Types are extended:

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

Two new typing rules are introduced:

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

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

Sums with unique types

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

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

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

and modifying the typing rules and reduction rules accordingly.

Exercise

Describe an extension with the option type.

Modularity of extensions

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

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

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

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

Modularity of extensions

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

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

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

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

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

Recursive functions

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

Values and expressions are extended:

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

A new reduction rule is introduced:

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

Recursive functions

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

What does this imply as a corollary?

— Types will not distinguish functions from recursive functions.

A new typing rule is introduced:

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

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

A derived construct: let

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

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

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

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

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

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

A derived construct: let or a primitive one?

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

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

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

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

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

A derived construct: let rec

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

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

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

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

Contents

What is polymorphism?

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

Why polymorphism?

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

In short, it should have polymorphic type:

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

which *instantiates* to the monomorphic types:

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

Why polymorphism?

In the absence of polymorphism, the only ways of achieving this effect would be:

- to manually duplicate the list sorting function at every type (*no-no!*);
- to use subtyping and claim that the function sorts lists of values of *any* type:

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

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

  Why isn't this so good? This leads to *loss of information* and subsequently requires introducing an unsafe *downcast* operation. This was the approach followed in Java before generics were introduced in 1.5.

Polymorphism seems almost free

Polymorphism is already implicitly present in simply-typed λ-calculus. Indeed, we have checked that the type:

$$(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

is a *principal type* for the term $\lambda f x y. (f\ x, f\ y)$.

By saying that this term admits the polymorphic type:

$$\forall \alpha_1 \alpha_2. (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

we make polymorphism *internal* to the type system.

Towards type abstraction

Polymorphism is a step on the road towards *type abstraction.*

Intuitively, if a function that sorts a list has polymorphic type:

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

then it *knows nothing* about α—it is *parametric* in α—so it must manipulate the list elements *abstractly:* it can copy them around, pass them as arguments to the comparison function, but it cannot directly inspect their structure.

In short, within the code of the list sorting function, the variable α is an *abstract type*.

Parametricity

In the presence of polymorphism (and in the absence of effects), a type can reveal a lot of information about the terms that inhabit it.

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

Similarly, the type of the list sorting function

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

reveals a *"free theorem"* about its behavior!

Basically, sorting commutes with (map f), provided f is order-preserving.

$$(\forall x, y, cmp\;(f\;x)\;(f\;y) = cmp\;x\;y) \implies \forall \ell, sort\;(map\;f\;\ell) = map\;f\;(sort\;\ell)$$

Note that there are many inhabitants of this type, but they all satisfy this free theorem (including, e.g., a function that sorts in reverse order, or a function that removes duplicates)

Parametricity

In the presence of polymorphism (and in the absence of effects), a type can reveal a lot of information about the terms that inhabit it.

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

Similarly, the type of the list sorting function

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

reveals a *"free theorem"* about its behavior!

Basically, sorting commutes with (map f), provided f is order-preserving.

$$(\forall x, y, cmp\ (f\ x)\ (f\ y) = cmp\ x\ y) \implies \forall \ell, sort\ (map\ f\ \ell) = map\ f\ (sort\ \ell)$$

Note that there are many inhabitants of this type, but they all satisfy this free theorem (including, e.g., a function that sorts in reverse order, or a function that removes duplicates)

Ad hoc *v.s.* *parametric* polymorphism

The term "polymorphism" dates back to a 1967 paper by Strachey [2000], where *ad hoc polymorphism* and *parametric polymorphism* were distinguished.

There are two different (and sometimes incompatible) ways of defining this distinction...

Ad hoc *v.s.* parametric polymorphism: first definition

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

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

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

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

Ad hoc *v.s.* parametric polymorphism: second definition

With parametric polymorphism, *untyped programs have a well-defined semantics.* (Think of the identity function.) Types are used only to rule out unsafe programs.

With ad hoc polymorphism, untyped programs do not have a semantics: *the meaning of a term can depend upon its type* (e.g. $2 + 2$), or, even worse, *upon its type derivation* (e.g. $\lambda x. \, show \; (read \, x)$).

Ad hoc *v.s.* parametric polymorphism: type classes

By the first definition, Haskell's *type classes* [Hudak et al., 2007] are a form of (bounded) parametric polymorphism: terms have *principal (qualified) type schemes*, such as:

$$\forall \alpha. \mathit{Num}\, \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

Yet, by the second definition, type classes are a form of ad hoc polymorphism: untyped programs do not have a semantics.

In the case of Haskell type classes, the two views can be reconciled. (See the course on overloading.)

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

Contents

System F

The System F, (also known as: the *polymorphic* λ-calculus, the *second-order* λ-calculus; F^2) was independently defined by Girard (1972) and Reynolds [1974].

Compared to the simply-typed λ-calculus, types are extended with universal quantification:

$$\tau ::= \ldots \mid \forall \alpha . \tau$$

How are the **syntax** and **semantics** of terms extended?

There are several variants, depending on whether one adopts an

- *implicitly-typed* or *explicitly-typed* (syntactic) presentation of terms
- and a *type-passing* or a *type-erasing* semantics.

Explicitly-typed System F

In the explicitly-typed variant [Reynolds, 1974], there are term-level constructs for introducing and eliminating the universal quantifier:

$$\textsc{Tabs}\;\frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\tau} \qquad \textsc{Tapp}\;\frac{\Gamma \vdash M : \forall\alpha.\tau}{\Gamma \vdash M\;\tau' : [\alpha \mapsto \tau']\tau}$$

Terms are extended accordingly:

$$M ::= \ldots \mid \Lambda\alpha.\, M \mid M\;\tau$$

Type variables are explicitly bound and appear in type environments.

$$\Gamma ::= \ldots \mid \Gamma, \alpha$$

Well-formedness of environment

Mandatory: We extend our previous convention to form environments: Γ, α requires $\alpha \# \Gamma$, *i.e.* α is neither in the domain nor in the image of Γ.

Optional: We also require that environments be closed with respect to type variables, that is, we require $\mathrm{ftv}(\tau) \subseteq \mathrm{dom}(\Gamma)$ to form $\Gamma, x : \tau$.

However, a looser style would also be possible.

- Our stricter definition allows fewer judgments, since judgments with open contexts are not allowed.
- However, these judgments can always be closed by adding a prefix composed of a sequence of its free type variables to be well-formed.

The stricter presentation is easier to manipulate in proofs;
it is also easier to mechanize.

Well-formedness of environments and types

Well-formedness of environments, written $\vdash \Gamma$ and well-formedness of types, written $\Gamma \vdash \tau$, may also be defined *recursively* by inference rules:

$$\textsc{WfEnv-Empty}\quad \frac{}{\vdash \varnothing} \qquad \textsc{WfEnvTvar}\quad \frac{\vdash \Gamma \qquad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha} \qquad \textsc{WfEnvVar}\quad \frac{\Gamma \vdash \tau \qquad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : \tau}$$

$$\textsc{WfTypeVar}\quad \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{WfTypeArrow}\quad \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \qquad \textsc{WfTypeForall}\quad \frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha. \tau}$$

Note

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

Well-formedness of environments and types

There is a choice whether well-formedness of environments should be made explicit or left implicit in typing rules.

Explicit well-formedness amounts to adding well-formedness premises to every rule where the environment or some type that appears in the conclusion does not appear in any premise.

$$\textsc{Var}\ \frac{x : \tau \in \Gamma \qquad \boxed{\vdash \Gamma}}{\Gamma \vdash x : \tau} \qquad \qquad \textsc{Tapp}\ \frac{\Gamma \vdash M : \forall \alpha. \tau \qquad \boxed{\Gamma \vdash \tau'}}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}$$

Explicit well-formedness is more precise and better suited for mechanized proofs. Explicit well-formedness is recommended.

However, we choose to leave well-formedness conditions implicit in this course, as it is a bit verbose and sometimes distracting. *(Still, we will remind implicit well-formedness premises in the definition of typing rules.)*

Type-passing semantics

We need the following reduction for type-level expressions:

$$(\Lambda\alpha.\, M)\; \tau \longrightarrow [\alpha \mapsto \tau]M \qquad (\iota)$$

Then, there is a choice.

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

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

However, this defines a type-passing semantics!

Indeed, $\Lambda\alpha.\, ((\lambda y : \alpha.\, y)\; V)$ is then a value while its type erasure $(\lambda y.\, y)\; \lceil V \rceil$ is not—and can be further reduced.

Type-erasing semantics

We recover a type-erasing semantics if we allow evaluation under type abstraction:

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

Then, we only need a weaker version of ι-reduction:

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

We now have:

$$\Lambda\alpha. ((\lambda y : \alpha. y)\ V) \longrightarrow \Lambda\alpha. V$$

We verify below that this defines a type-erasing semantics, indeed.

Type-passing versus type-erasing: pros and *cons*

The type-passing interpretation has a number of disadvantages.

- because it alters the semantics, it does not fit our view that *the untyped semantics should pre-exist* and that a type system is only a predicate that selects a subset of the well-behaved terms.

- it blocks reduction of polymorphic expressions:

  if f is list flattening of type $\forall \alpha.\, list\,(list\,\alpha) \rightarrow list\,\alpha$, the monomorphic function $(f\ int) \circ (f\ (list\ int))$ reduces to $\Lambda x.\, f\ (f\ x)$, while its more general polymorphic version $\Lambda \alpha.\,(f\ \alpha) \circ (f\ (list\,\alpha))$ is irreducible.

- because it requires both values and types to exist at runtime, it can lead to a *duplication of machinery*. Compare type-preserving closure conversion in type-passing [Minamide et al., 1996] and in type-erasing [Morrisett et al., 1999] styles.

Type-passing versus type-erasing: *pros* and cons

An apparent advantage of the type-passing interpretation is to allow *typecase*; however, typecase can be simulated in a type-erasing system by viewing runtime *type descriptions* as *values* [Crary et al., 2002].

The *type-erasing* semantics

- does not alter the semantics of untyped terms.
- *for this very reason*, it also coincides with the semantics of ML—and, more generally, with the semantics of most programming languages.
- It also exhibits difficulties when adding side effects while the type-passing semantics does not.

In the following, we choose a type-erasing semantics.

Notice that we allow evaluation under a type abstraction as a consequence of choosing a type-erasing semantics—and not the converse.

Reconciling type-passing and type-erasing views

If we restrict type abstraction to value-forms (which include values and variables), that is, we only allow $\Lambda\alpha.\,M$ when M is a value-form, then the type-passing and type-erasing semantics coincide.

Indeed, under this restriction, closed type abstractions will always be type abstractions of values, and evaluation under type abstraction will never be used, even if allowed.

This restriction is chosen when adding side-effects as a way to preserve type-soundness.

Explicitly-typed System F

We study the *explicitly-typed* presentation of System F first because it is simpler.

Once, we have verified that the semantics is indeed type-preserving, many properties can be *transferred back* to the *implicitly-typed* version, and in particular, to its ML subset.

Then, both presentations can be used, interchangeably.

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

Syntax

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

Typing rules

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

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

$$\textsc{TAbs}\quad \frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda \alpha.\, M : \forall \alpha. \tau}$$

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

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

Semantics

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

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

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

Encoding data-structures

System F is quite expressive: it enables the *encoding* of data structures.

For instance, the church encoding of pairs is well-typed:

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

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

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

Primitive data-structures as constructors and destructors

Unit, Pairs, Sums, *etc.* can also be added to System F *as primitives*.

We can then proceed as for simply-typed λ-calculus.

However, we may take advantage of the expressiveness of System F to deal with such extensions in a more elegant way: thanks to polymorphism, we need not add new typing rules for each extension.

We may instead add one typing rule for constants that is parametrized by an initial typing environment.

This allows sharing the meta-theoretical developments between the different extensions.

Let us first illustrate an extension of System F with primitive pairs.
(We will then generalize it to arbitrary constructors and destructors.)

Constructors and destructors

Pairs

Types are extended with a type constructor $\times$ of arity 2:

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

Expressions are extended with a constructor $(\cdot, \cdot)$ and two destructors *proj*$_1$ and *proj*$_2$ with the respective signatures:

$$\begin{aligned} Pair : &\quad \forall \alpha_1. \forall \alpha_2. \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2 \\ proj_i : &\quad \forall \alpha_1. \forall \alpha_2. \alpha_1 \times \alpha_2 \to \alpha_i \end{aligned}$$

which represent an initial environment Δ. We need not add any new typing rule, but instead type programs in the initial environment Δ.

This allows for the formation of partial applications of constructors and destructors (all cases but one). Hence, values are extended as follows:

$$\begin{aligned} V \quad ::= \quad \ldots \quad & \mid Pair \mid Pair\ \tau \mid Pair\ \tau\ \tau \mid Pair\ \tau\ \tau\ V \mid Pair\ \tau\ \tau\ V\ V \\ & \mid proj_i \mid proj_i\ \tau \mid proj_i\ \tau\ \tau \end{aligned}$$

Constructors and destructors

Pairs

We add the two following reduction rules:

$$proj_i\ \tau_1\ \tau_2\ (pair\ \tau'_1\ \tau'_2\ V_1\ V_2) \longrightarrow V_i \qquad (\delta_{pair})$$

Comments?

- For well-typed programs, τ_i and τ'_i will always be equal, but the reduction will not check this at runtime.

  Instead, one could have defined the rule:

  $$proj_i\ \tau_1\ \tau_2\ (pair\ \tau_1\ \tau_2\ V_1\ V_2) \longrightarrow V_i \qquad (\delta'_{pair})$$

  The two semantics are equivalent on well-typed terms, but differ on ill-typed terms where δ'_{pair} may block when rule δ_{pair} would progress, ignoring type errors.

  Interestingly, with δ'_{pair}, the proof obligation is simpler for subject reduction but replaced by a stronger proof obligation for progress.

Constructors and destructors — Pairs

We add the two following reduction rules:

$$proj_i\ \tau_1\ \tau_2\ (pair\ \tau_1'\ \tau_2'\ V_1\ V_2) \longrightarrow V_i \qquad (\delta_{pair})$$

Comments?

- This presentation forces the programmer to specify the types of the components of the pair.

  However, since this is an explicitly type presentation, these types are already known from the arguments of the pair (when present)

  This should not be considered as a problem: explicitly-typed presentations are always verbose. Removing redundant type annotations is the task of type reconstruction.

Constructors and destructors — General case

Assume given a collection of type constructors $G \in \mathcal{G}$, with their arity *arity* (G). We assume that types respect the arities of type constructors.

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

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

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

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

We require that

- $\boxed{\tau}$ be a datatype whenever c is a constructor (key for progress);
- the arity of destructors be strictly positive
  (nullary destructors introduce pathological cases for little benefit).

Constructors and destructors — General case

Expressions are extended with constants: Constants are typed as variables, but their types are looked up in the initial environment Δ:

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

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

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

For each destructor d of arity n, we assume given a set of δ-rules of the form

$$d\ \tau_1\ \ldots\ \tau_k\ V_1\ \ldots\ V_n \longrightarrow M \qquad (\delta_d)$$

Constructors and destructors — Soundness requirements

Of course, we need assumptions to relate typing and reduction of constants:

Subject-reduction for constants:

- δ-rules preserve typings for well-typed terms

  If $\vec{\alpha} \vdash M_1 : \tau$ and $M_1 \longrightarrow_\delta M_2$ then $\vec{\alpha} \vdash M_2 : \tau$.

Progress for constants:

- Well-typed full applications of destructors can be reduced

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

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

Example Unit

Adding units:

- Introduce a type constant *unit*
- Introduce a constructor $()$ of arity 0 of type *unit*.
- No primitive and no reduction rule is added.

The assumptions obviously hold in the absence of destructors.

The previous example of pairs also perfectly fits in this framework.

Example Fixpoint

We introduce a destructor

$$\mathit{fix} : \forall\alpha.\,\forall\beta.\,((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta \qquad \in \Delta$$

of arity 2, together with the δ-rule

$$\mathit{fix}\;\tau_1\;\tau_2\;V_1\;V_2 \longrightarrow V_1\;(\mathit{fix}\;\tau_1\;\tau_2\;V_1)\;V_2 \qquad (\delta_{\mathit{fix}})$$

It is straightforward to check the assumptions:

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

  (by inspection of the typing derivation)

  Assume that $\Gamma \vdash \mathit{fix}\;\tau_1\;\tau_2\;V_1\;V_2 : \tau$. By inversion of typing rules, τ must be equal to τ_2, V_1 and V_2 must be of types $(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$ and τ_1 in the typing context Γ. We may then easily build a derivation of the judgment $\Gamma \vdash V_1\;(\mathit{fix}\;\tau_1\;\tau_2\;V_1)\;V_2 : \tau$

Exercise

Lists

1) Formulate the extension of System F with lists as constants.

2) Check that this extension is sound.

Solution

1) We introduce a new unary type constructor *list* ; two constructors *Nil* $\cdot$ and *Cons* of types $\forall\alpha.\,\mathit{list}\,\alpha$ and $\forall\alpha.\,\alpha \to \mathit{list}\,\alpha \to \mathit{list}\,\alpha$; and one destructor *matchlist* $\cdot \cdot \cdot\cdot$ of type:

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

with the two reduction rules:

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

2) See the case of pairs in the course.

Contents

Type soundness

The structure of the proof is similar to the case of simply-typed λ-calculus and follows from subject reduction and progress.

Subject reduction uses the following lemmas:

- inversion of typing judgments
- permutation and weakening
- expression substitution
- type substitution (new)
- compositionality

Inversion of typing judgements

Lemma (Inversion of typing rules)

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

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

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

Type soundness — Weakening

Lemma (Weakening)

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

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

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

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

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

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

Lemma (Permutation)

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

Type soundness

Type substitution

Lemma (Expression substitution, *strengthened*)

If $\Gamma, x : \tau_0, \Gamma' \vdash M : \tau$ *and* $\Gamma \vdash M_0 : \tau_0$ *then* $\Gamma, \Gamma' \vdash [x \mapsto M_0]M : \tau$.

The proof is by induction on M.

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

Type soundness Type substitution

Lemma (Type substitition, strengthened)

If $\Gamma, \alpha, \Gamma' \vdash M : \tau'$ *and* $\Gamma \vdash \tau$ *then* $\Gamma, [\alpha \mapsto \tau]\Gamma' \vdash [\alpha \mapsto \tau]M : [\alpha \mapsto \tau]\tau'$.

The proof is by induction on M.

The interesting cases are for type and value abstraction, which require the strengthened version with an arbitrary typing context Γ' on the right. Then, the proof is straightforward.

Compositionality

Lemma (Compositionality)

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

Remarks

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

Type soundness — Subject reduction

Theorem (Subject reduction)

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

The proof is by induction on M.
Using the previous lemmas it is straightforward.

Interestingly, the case for δ-rules follows from the subject-reduction assumption for constants (slide 78).

Type soundness — Progress

Progress is restated as follows:

Theorem (Progress, strengthened)

A well-typed, irreducible closed term is a value:
if $\vec{\alpha} \vdash M : \tau$ *and* $M \not\longrightarrow$ *, then* M *is some value* V*.*

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

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

It relies mainly on the *classification lemma* (given below) and the progress assumption for destructors (slide 78).

Type soundness — Classification

Beware! We must take care of partial applications of constants

Lemma (Classification)

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

- *If τ is an arrow type, then V is either a function or a partial application of a constant.*
- *If τ is a polymorphic type, then V is either a type abstraction of a value or a partial application of a constant to types.*
- *If τ is a constructed type, then V is a constructed value.*

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

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

Normalization

Theorem

Reduction terminates in pure System F.

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

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

See the lesson on logical relations.

Contents

Implicitly-typed System F

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

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

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

Definition 2 (equivalent) Typing rules for $\lceil F \rceil$ are those of the implicitlty-typed simply-typed λ-calculus with two new rules:

$$\textsc{if-Tabs}\ \frac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall \alpha . \tau} \qquad \textsc{if-Tapp}\ \frac{\Gamma \vdash a : \forall \alpha . \tau}{\Gamma \vdash a : [\alpha \mapsto \tau_0]\tau}$$

Notice that these rules are not syntax directed.

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

Notice that the explicit introduction of variable α in the premise of Rule Tabs contains an implicit side condition $\alpha \# \Gamma$ due to the global assumption on the formation of Γ, α:

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

In implicitly-typed System F, we could also omit type declarations from the typing environment. (Although, in some extensions of System F, type variables may carry a kind or a bound and must be explicitly introduced.)

Then, we would need an explicit side-condition as in if-Tabs-Bis:

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

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

Omitting the side condition leads to *unsoundness:*

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

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

This is a type derivation for a *type cast* (Objective Caml's Obj.magic).

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

This is equivalent to using an ill-formed typing environment :

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

$\alpha_1, \alpha_2, x : \alpha_1, \alpha_1$ ill-formed

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

A good intuition is: a judgment $\Gamma \vdash a : \tau$ corresponds to the logical assertion $\forall \vec{\alpha}.(\Gamma \Rightarrow \tau)$, where $\vec{\alpha}$ are the free type variables of the judgment.

In that view, TABS-BIS corresponds to the axiom:

$$\forall \alpha.(P \Rightarrow Q) \quad \equiv \quad P \Rightarrow (\forall \alpha.Q) \qquad \text{if } \alpha \# P$$

Type-erasing typechecking

Type systems for implicitly-typed and explicitly-type System F coincide.

Lemma

$\Gamma \vdash a : \tau$ holds in implicitly-typed System F if and only if there exists an explicitly-typed expression M whose erasure is a such that $\Gamma \vdash M : \tau$.

Trivial.

One could write judgements of the form $\Gamma \vdash a \Rightarrow M : \tau$ to mean that the *explicitly typed* term M witnesses that the *implicitly typed* term a has type τ in the environment Γ.

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

Here is a version of the term $\lambda fxy.(f\ x, f\ y)$ that carries explicit type abstractions and annotations:

$$\Lambda\alpha_1.\Lambda\alpha_2.\lambda f : \alpha_1 \to \alpha_2.\lambda x : \alpha_1.\lambda y : \alpha_1.(f\ x, f\ y)$$

This term admits the polymorphic type:

$$\forall\alpha_1.\forall\alpha_2.(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

Quite unsurprising, right? Perhaps more surprising is the fact that this untyped term can be decorated in a different way:

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

This term admits the polymorphic type:

$$\forall\alpha_1.\forall\alpha_2.(\forall\alpha.\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

This begs the question: ...

Incomparable types in System F $\lambda f x y.\,(f\ x, f\ y)$

Which of the two is more general?

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

The first one requires x and y to admit a common type, while the second one requires f to be polymorphic.

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

Take, for instance,

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

and

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

Distrib pair in F^ω (parenthesis)

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

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

call it "dp" of type:

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

Then

$$\begin{array}{l} \mathsf{dp}\ (\lambda\alpha.\alpha)(\lambda\alpha.\alpha) \\ \qquad : \forall\alpha_1.\forall\alpha_2.(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2 \end{array}$$

$$\begin{array}{l} \Lambda\alpha_1.\Lambda\alpha_2.\ \mathsf{d}p\ (\lambda\alpha.\alpha_1)\ (\lambda\alpha.\alpha_2)\ \alpha_1\ \alpha_2 \\ \qquad : \forall\alpha_1.\forall\alpha_2.(\forall\alpha.\,\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2 \end{array}$$

Notions of instance in $\lceil F \rceil$

It seems plausible that the untyped term $\lambda fxy.(f\ x, f\ y)$ does not admit a type τ_0 of which the two previous types are instances.

But, in order to prove this, one must fix what it means for τ_2 to be an *instance* of τ_1—or, equivalently, for τ_1 to be *more general* than τ_2.

Several definitions are possible...

Syntactic notions of instance in $\lceil F \rceil$

In System F, *to be an instance* is usually defined by the rule:

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

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

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

Perhaps surprisingly, the rule is *not derivable* in our presentation of System F as the proof of admissibility requires weakening.
(It would be derivable if we had left type variables implicit in contexts.)

Syntactic notions of instance in F

What is the counter-part of instance in explicitly-typed System F?

Assume $\Gamma \vdash M : \tau_1$ and $\tau_1 \leq \tau_2$. How can we see M with type τ_2?

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

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

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

Admissible rule:

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

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

Retyping contexts in F

In F, subtyping is a judgment $\Gamma \vdash \tau_1 \leq \tau_2$, rather than a binary relation, where the context Γ keeps track of well-formedness of types. Subtyping relations can be witnessed by retyping contexts.

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

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

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

Let us write $\Gamma \vdash \mathcal{R}[\tau_1] : \tau_2$ iff $\Gamma, x : \tau_1 \vdash \mathcal{R}[x] : \tau_2$ (where $x \# \mathcal{R}$)

If $\Gamma \vdash M : \tau_1$ and $\Gamma \vdash \mathcal{R}[\tau_1] : \tau_2$, then $\Gamma \vdash \mathcal{R}[M] : \tau_2$,

Then $\Gamma \vdash \tau_1 \leq \tau_2$ iff $\Gamma \vdash \mathcal{R}[\tau_1] : \tau_2$. for some retyping context $\mathcal{R}$.

In System F, retyping contexts can only change *toplevel* polymorphism: they cannot operate under arrow types to weaken the return type or strengthen the domain of functions.

Another syntactic notion of instance: F_η

Mitchell [1988] defined F_η, a version of $\lceil F \rceil$ extended with a richer *instance* relation as:

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

$$\textsc{Distributivity}\quad \forall \alpha.\,(\tau_1 \to \tau_2) \leq (\forall \alpha.\,\tau_1) \to (\forall \alpha.\,\tau_2)$$

$$\textsc{Congruence-}\!\to\quad \frac{\tau_2 \leq \tau_1 \qquad \tau_1' \leq \tau_2'}{\tau_1 \to \tau_1' \leq \tau_2 \to \tau_2'}$$

$$\textsc{Congruence-}\forall\quad \frac{\tau_1 \leq \tau_2}{\forall \alpha.\tau_1 \leq \forall \alpha.\tau_2}$$

$$\textsc{Transitivity}\quad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

In F_η, Rule Sub must be primitive as it is not admissible (but still sound).

F_η can also be defined as the closure of System F under η-equality.

Why is a rich notion of instance potentially interesting?

- More polymorphism.
- More hope of having principal types.

A definition of principal typings

A typing of an expression M is a pair Γ, τ such that $\Gamma \vdash M : \tau$.

Ideally, a type system should have *principal typings* [Wells, 2002]:

> *Every well-typed term M admits a principal typing – one whose instances are exactly the typings of M.*

Whether this property holds depends on a definition of *instance*. The more liberal the instance relation, the more hope there is of having principal typings.

A *semantic* notion of instance

Wells [2002] notes that, once a type system is fixed, a most liberal notion of instance can be defined, a posteriori, by:

> *A typing θ_1 is more general than a typing θ_2 if and only if every term that admits θ_1 admits θ_2 as well.*

This is the largest reasonable notion of instance: $\leq$ is defined as the largest relation such that a subtyping principle (for typings) is admissible.

This definition can be used to prove that a system does *not* have principal typings, under *any* reasonable definition of "instance".

Which systems have principal typings?

The *simply-typed λ-calculus has principal typings,* with respect to a substitution-based notion of instance. (See course notes on type inference.)

Wells [2002] shows that *neither System F nor F_η have principal typings.*

It was shown earlier that *F_η's instance relation is undecidable* [Wells, 1995; Tiuryn and Urzyczyn, 2002] and that *type inference for both System F and F_η is undecidable* [Wells, 1999].

Which systems have principal typings?

There are still a few positive results...

Some systems of *intersection types* have principal typings [Wells, 2002] – but they are very complex and have yet to see a practical application.

A weaker property is to have *principal types*. Given an environment Γ and an expression M, is there a type τ for M in Γ such that all other types of M in Γ are instances of τ.

Damas and Milner's type system (coming up next) does not have *principal typings* but it has *principal types* and *decidable type inference.*

Other approaches to type inference in System F

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

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

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

Type soundness for $\lceil F \rceil$

Subject reduction and progress imply the soundness of the *explicitly*-typed System F. What about the *implicitly*-typed version?

Can we reuse the soundness proof for the explicitly-typed version? Can we pull back subject reduction and progress from F to $\lceil F \rceil$?

Progress? Given a well-typed term $a \in \lceil F \rceil$, can we find a term $M \in F$ whose erasure is a and since M is a value or reduces, conclude that a is a value or reduces?

Subject reduction? Given a well-typed term $a_1 \in \lceil F \rceil$ of type τ that reduces to a_2, can we find a term $M_1 \in F$ whose erasure is a_1 and show that M_1 reduces to a term M_2 whose erasure is a_2 to conclude that the type of a_2 is the same as the type of a_1?

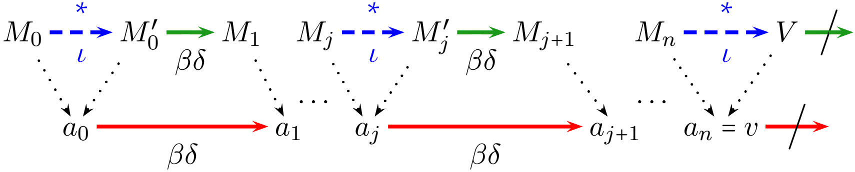
In both cases, this reasoning requires a *type-erasing* semantics.

Type erasing semantics

We claimed earlier that the explicitly-typed System F has an erasing semantics. We now verify it.

There is a difference with the simply-typed λ-calculus because the reduction of type applications on explicitly-typed terms is dropped on implicitly-typed terms, hence the two reductions cannot coincide *exactly*.

The way to formalize this is to split reduction steps into $\beta\delta$-steps corresponding to β or δ rules that are preserved by type-erasure, and ι-steps corresponding to the reduction of type applications that disappear during type-erasure:



Type erasing semantics — Direct simulation

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

Lemma (Direct simulation)

Assume $\Gamma \vdash M_1 : \tau$.
1) If $M_1 \longrightarrow_\iota M_2$, *then* $\lceil M_1 \rceil = \lceil M_2 \rceil$
2) If $M_1 \longrightarrow_{\beta\delta} M_2$, *then* $\lceil M_1 \rceil \longrightarrow_{\beta\delta} \lceil M_2 \rceil$

Both parts are easy by definition of type erasure.

Type erasing semantics Inverse simulation

The inverse direction is more delicate to state, since there are usually many expressions of F whose erasure is a given expression in $\lceil F \rceil$, as $\lceil \cdot \rceil$ is not injective.

Lemma (Inverse simulation)

Assume $\Gamma \vdash M_1 : \tau$ *and* $\lceil M_1 \rceil \longrightarrow a$.
Then, there exists a term M_2 *such that* $M_1 \longrightarrow_\iota^* \longrightarrow_{\beta\delta} M_2$ *and* $\lceil M_2 \rceil = a$.

Type erasing semantics — Assumption on δ-reduction

Of course, the semantics can only be type erasing if δ-rules do not themselves depend on type information.

We first need δ-reduction to be defined on type erasures.

- We may prove the theorem directly for some concrete examples of δ-reduction.
  However, keeping δ-reduction abstract is preferable to avoid repeating the same reasoning again and again.
- We assume that it is such that type erasure establishes a bisimulation for δ-reduction taken alone.

Type erasing semantics — Assumption on δ-reduction

We assume that for any explicitly-typed term M of the form $d\,\tau_1 \ldots \tau_j\, V_1 \ldots V_k$ such that $\Gamma \vdash M : \tau$, the following properties hold:

(1) If $M \longrightarrow_\delta M'$, then $\lceil M \rceil \longrightarrow_\delta \lceil M' \rceil$.

(2) If $\lceil M \rceil \longrightarrow_\delta a$, then there exists M' such that $M \longrightarrow_\delta M'$ and a is the type-erasure of M'.

Remarks

- In most cases, the assumption on δ-reduction is obvious to check.
- In general the δ-reduction on untyped terms is larger than the projection of δ-reduction on typed terms.
- If we restrict δ-reduction to implicitly-typed terms, then it usually coincides with the projection of δ-reduction of explicitly-typed terms.

Type soundness for implicitly-typed System F

We may now easily transpose subject reduction and progress from the implicitly-typed version to the implicitly-typed version of System F.

Progress Well-typed expressions in $\lceil F \rceil$ have a well-typed antecedent in ι-normal form in F, which, by progress in F, either $\beta\delta$-reduces or is a value; then, its type erasure $\beta\delta$-reduces (by direct simulation) or is a value (by observation).

Subject reduction Assume that $\Gamma \vdash a_1 : \tau$ and $a_1 \longrightarrow a_2$.

- By well-typedness of a_1, there exists a term M_1 that erases to a_1 such that $\Gamma \vdash M_1 : \tau$.
- By inverse simulation in F, there exists M_2 such that $M_1 \longrightarrow_\iota^* \longrightarrow_{\beta\delta} M_2$ and $\lceil M_2 \rceil$ is a_2.
- By subject reduction in F, $\Gamma \vdash M_2 : \tau$, which implies $\Gamma \vdash a_2 : \tau$.

Type erasing semantics

The design of advanced typed systems for programming languages is usually done in explicitly-typed versions, with a type-erasing semantics in mind, but this is not always checked in details.

While the direct simulation is usually straightforward, the inverse simulation is often harder. As type systems get more complicated, reduction at the level of types also gets more complicated.

It is important and not always obvious that **type reduction** *terminates and is rich enough to never block reductions that could occur in the type erasure.*

Type erasing semantics — On bisimulations

Using bisimulations to show that compilation preserves the semantics given in small-step style is a classical technique.

For example, this technique is *heavily* used in the CompCert project to prove the correctness of a C-compiler to assembly code in Coq, using a dozen of successive intermediate languages.

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

Proof of inverse simulation

The inverse simulation can first be shown assuming that M_1 is ι-normal.

The general case follows, since then M_1 ι-reduces to a normal form M_1' preserving typings; then, the lemma can be applied to M_1' instead of M_1.

Notice that this argument relies on the termination of ι-reduction alone.

The termination of ι-reduction is easy for System F, since it strictly decreases the number of type abstractions. (In F^ω, it requires termination of simply-typed λ-calculus.)

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

Proof of inverse simulation — Helper lemmas

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

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

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

Lemma

1) *A term that erases to $\bar{e}[a]$ can be put in the form $\bar{E}[M]$ where $\lceil \bar{E} \rceil$ is $\bar{e}$ and $\lceil M \rceil$ is a, and moreover, M does not start with a type abstraction nor a type application.*
2) *An evaluation context $\bar{E}$ whose erasure is the empty context is a retyping context $\mathcal{R}$.*
3) *If $\mathcal{R}[M]$ is in ι-normal form, then $\mathcal{R}$ is of the form $\Lambda\vec{\alpha}.[]\;\vec{\tau}$.*

Proof of inverse simulation — Helper lemmas

Lemma (inversion of type erasure)

Assume $\lceil M \rceil = a$

- *If* a *is* x, *then* M *is of the form* $\mathcal{R}[x]$
- *If* a *is* c, *then* M *is of the form* $\mathcal{R}[c]$
- *If* a *is* $\lambda x.\, a_1$, *then* M *is of the form* $\mathcal{R}[\lambda x{:}\tau.\, M_1]$ *with* $\lceil M_1 \rceil = a_1$
- *If* a *is* $a_1\ a_2$, *then* M *is of the form* $\mathcal{R}[M_1\ M_2]$ *with* $\lceil M_i \rceil = a_i$

The proof is by induction on M.

Proof of inverse simulation — Helper lemmas

Lemma (Inversion of type erasure for well-typed values)

Assume $\Gamma \vdash M : \tau$ and M is ι-normal. If $\lceil M \rceil$ is a value v, then M is a value V. Moreover,

- *If v is $\lambda x. a_1$, then V is $\Lambda \vec{\alpha}. \lambda x{:}\tau. M_1$ with $\lceil M_1 \rceil = a_1$.*
- *If v is a partial application $c\, v_1 \ \ldots v_n$ then V is $\mathcal{R}[c\, \vec{\tau}\, V_1 \ \ldots V_n]$ with $\lceil V_i \rceil = v_i$.*

The proof is by induction on M. It uses the inversion of type erasure and analysis of the typing derivation to restrict the form of retyping contexts.

Corollary

Let M be a well-typed term in ι-normal form whose erasure is a.

- *If a is $(\lambda x. a_1)\, v$, then M if of the form $\mathcal{R}[(\lambda x{:}\tau. M_1)\, V]$, with $\lceil M_1 \rceil = a_1$ and $\lceil V \rceil = v$.*
- *If a is a full application $(d\, v_1 \ldots v_n)$, then M is of the form $\mathcal{R}[d\, \vec{\tau}\, V_1 \ \ldots V_n]$ and $\lceil V_i \rceil$ is v_i.* $\square$

Abstract Data types, Existential types, GADTs

Contents

Algebraic Datatypes Types — Examples

In OCaml:

```
type 'a list =
  | Nil : 'a list
  | Cons : 'a * 'a list → 'a list
```

or

```
type ('leaf, 'node) tree =
  | Leaf : 'leaf → ('leaf, 'node) tree
  | Node : ('leaf, 'node) tree * 'node * ('leaf, 'node) tree → ('leaf, 'node) tree
```

Algebraic Datatypes Types — General case

General case

$$\text{type } G\,\vec{\alpha} = \Sigma_{i\in 1..n}(C_i : \forall\vec{\alpha}.\,\tau_i \to G\,\vec{\alpha}) \qquad \text{where } \vec{\alpha} = \bigcup_{i\in 1..n} \mathrm{ftv}(\tau_i)$$

In System F, this amounts to declaring:

- a new type constructor G,
- n constructors $\quad C_i \;:\; \forall\vec{\alpha}.\,\tau_i \to G\,\vec{\alpha}$
- one destructor $\quad d_G \;:\; \forall\vec{\alpha},\gamma.\,G\,\vec{\alpha} \to (\tau_1 \to \gamma)\ldots(\tau_n \to \gamma) \to \gamma$
- n reduction rules $\quad d_G\,\bar{\tau}\,(C_i\,\bar{\tau}'\,v)\,v_1\,\ldots v_n \;\rightsquigarrow\; v_i\,v$

Exercise

Show that this extension verifies the subject reduction and progress axioms for constants.

Algebraic Datatypes Types

General case

$$\text{type } G\,\vec{\alpha} = \Sigma_{i\in 1..n}(C_i : \forall\vec{\alpha}.\,\tau_i \to G\,\vec{\alpha}) \qquad \text{where } \vec{\alpha} = \bigcup_{i\in 1..n} \mathrm{ftv}(\tau_i)$$

Notice that

- All constructors build values of the same type $G\ \vec{\alpha}$ and are surjective (all types can be reached)
- The definition may be recursive, *i.e.* G may appear in τ_i

Algebraic datatypes introduce *isorecursive types*.

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Recursive Types

Product and sum types alone do not allow describing *data structures* of *unbounded size*, such as lists and trees.

Indeed, if the grammar of types is $\tau ::= \mathit{unit} \mid \tau \times \tau \mid \tau + \tau$, then it is clear that every type describes a *finite* set of values.

For every k, the type of lists of length at most k is expressible using this grammar. However, the type of lists of unbounded length is not.

Equi- versus isorecursive types

The following definition is inherently *recursive:*

> *"A list is either empty or a pair of an element and a list."*

We need something like this:

$$\mathit{list}\ \alpha \quad \diamond \quad \mathit{unit} + \alpha \times \mathit{list}\ \alpha$$

But what does $\diamond$ stand for? Is it *equality,* or some kind of *isomorphism?*

There are two standard approaches to recursive types:

- *equirecursive* approach:
  a recursive type is *equal* to its unfolding.
- *isorecursive* approach:
  a recursive type and its unfolding are related via explicit *coercions*.

Equirecursive types

In the equirecursive approach, the usual syntax of types:

$$\tau ::= \alpha \mid \mathsf{F}\,\vec{\tau} \mid \forall\beta.\tau$$

is no longer interpreted inductively. Instead, types are the *regular infinite trees* built on top of this grammar.

Finite syntax for recursive types

$$\tau ::= \alpha \mid \mu\alpha.(\mathsf{F}\,\vec{\tau}) \mid \mu\alpha.(\forall\beta.\tau)$$

We do not allow the seemingly more general form $\mu\alpha.\tau$, because $\mu\alpha.\alpha$ is meaningless, and $\mu\alpha.\beta$ or $\mu\alpha.\mu\beta.\tau$ are useless. If we write $\mu\alpha.\tau$, it should be understood that τ is contractive, that is, τ is a type constructor application or a forall introduction.

For instance, the type of lists of elements of type α is:

$$\mu\beta.(\mathit{unit} + \alpha \times \beta)$$

Equirecursive types — Equality

Inductive definition [Brandt and Henglein, 1998] show that equality is the least congruence generated by the following two rules:

Fold/Unfold

$$\mu\alpha.\tau = [\alpha \mapsto \mu\alpha.\tau]\tau$$

Uniqueness

$$\frac{\tau_1 = [\alpha \mapsto \tau_1]\tau \qquad \tau_2 = [\alpha \mapsto \tau_2]\tau}{\tau_1 = \tau_2}$$

In both rules, τ must be contractive.

This axiomatization does not directly lead to an efficient algorithm for deciding equality, though.

Co-inductive definition

$$\alpha = \alpha \qquad \frac{[\alpha \mapsto \mu\alpha.\mathsf{F}\vec{\tau}]\vec{\tau} = [\alpha \mapsto \mu\alpha.\mathsf{F}\vec{\tau}']\vec{\tau}'}{\mu\alpha.\mathsf{F}\vec{\tau} = \mu\alpha.\mathsf{F}\vec{\tau}'} \qquad \frac{[\alpha \mapsto \mu\alpha.\forall\beta.\,\tau]\tau = [\alpha \mapsto \mu\alpha.\forall\beta.\,\tau']\tau'}{\mu\alpha.\forall\beta.\,\tau = \mu\alpha.\forall\beta.\,\tau'}$$

Equirecursive types Equality

In the absence of quantifiers

Each type in this syntax denotes a unique regular tree, sometimes known as its *infinite unfolding.* Conversely, every regular tree can be expressed in this notation (possibly in more than one way).

If one builds a type-checker on top of this finite syntax, then one must be able to *decide* whether two types are *equal,* that is, have identical infinite unfoldings.

This can be done efficiently, either via the algorithm for comparing two DFAs, or better, by unification. (The latter approach is simpler, faster, and extends to the type inference problem.)

Exercise

Show that $\mu\alpha.A\alpha = \mu\alpha.AA\alpha$ *and* $\mu\alpha.AB\alpha = A\mu\alpha.BA\alpha$ *with both inductive and co-inductive definitions. Can you do it without the* UNIQUENESS *rule?*

Equirecursive types

Without quantifiers

Proof of $\mu\alpha A A\alpha = \mu\alpha A A A\alpha$

By coinduction

Let | u be $\mu\alpha A A\alpha$
 | v be $\mu\alpha A A A\alpha$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{(1)}}{Au = Av}}{u = AAv}}{Au = v}}{u = Av}}{Au = AAv}}{u = v \;\; (1)}$$

By unification

Equivalent classes, using *small terms*	To do:
$u \sim Au_1 \;\wedge\; u_1 \sim Au \;\wedge\; v \sim Av_1 \;\wedge\; v_1 \sim Av_2 \;\wedge\; v_2 \sim Av$	$u \sim v$
$u \sim Au_1 \sim v \sim Av_1 \;\wedge\; u_1 \sim Au \;\wedge\; v_1 \sim Av_2 \;\wedge\; v_2 \sim Av$	$u_1 \sim v_1$
$u \sim v \sim Av_1 \;\wedge\; u_1 \sim Au \sim v_1 \sim Av_2 \;\wedge\; v_2 \sim Av$	$u \sim v_2$

Equirecursive types

In the presence of quantifiers

The situation is more subtle because of α-conversion.

A (somewhat involved) canonical form can still be found, so that checking equality and first-order unification on types can still be done in $O(n \log n)$. See [Gauthier and Pottier, 2004].

Otherwise, without the use of such canonical forms, the best known algorithm is in $O(n^2)$ [Glew, 2002] testing equality of automatons with binders.

Equirecursive types — With quantifiers

Example of unfolding with canonical forms [Gauthier and Pottier, 2004].

- the letter in green, is just any name, subject to α-conversion
- the number is the canonical name: it is the number of free variables under the binder—including recursive occurrences.

$$
\begin{aligned}
& \forall a1.\, \mu\ell.a1 \to \forall a2.\, (a2 \to \ell) && (1)\\
& \forall a1.\, \mu\ell.a1 \to \forall b2.\, (b2 \to \ell) && (\alpha)\\
= {} & \forall a1.\, \quad a1 \to \forall b2.\, (b2 \to \mu\ell.a1 \to \forall b2.\, (b2 \to \ell)) && (\mu)\\
= {} & \forall a1.\, \quad a1 \to \forall b2.\, (b2 \to \mu\ell.a1 \to \forall c2.\, (c2 \to \ell)) && (\alpha)
\end{aligned}
$$

With the canonical representation,

- Syntactic unfolding (*i.e.* without any renaming) avoids name capture and is also a correct semantical unfolding
- It shares free variables and can reuse the same name for the new bound variables without name capture.

Equirecursive types

Type soundness

In the presence of equirecursive types, structural induction on types is no longer permitted, but *we never used it* anyway – in soundness proofs.

We only need it to prove the termination of reduction, which does not hold any longer.

It remains true that

- $\mathsf{F}\ \vec{\tau}_1 = \mathsf{F}\ \vec{\tau}_2$ implies $\vec{\tau}_1 = \vec{\tau}_2$ (symbols are injective)—this is used in the proof of Subject Reduction.
- $\mathsf{F}_1\ \vec{\tau}_1 = \mathsf{F}_2\ \vec{\tau}_2$ implies $\mathsf{F}_1 = \mathsf{F}_2$—this is used in the proof of Progress.

So, the reasoning that leads to *type soundness* is unaffected.

Exercise

Prove type soundness for the simply-typed λ-calculus in Coq. Then, change the syntax of types from `Inductive` *to* `CoInductive`*.*

Equirecursive types break termination, indeed!

That is no a surprise, but...

What is the expressiveness of simply-typed λ-calculus with equirecursive types alone (no other constructs and/or constants)?

All terms of the untyped λ-calculus are typable!

- define the universal type U as $\mu\alpha.\alpha \to \alpha$
- we have $U = U \to U$, hence all terms are typable with type U.

Notce that one can emulate recursive types $U = U \to U$ by defining two functions *fold* and *unfold* of respective types $(U \to U) \to U$ and $U \to (U \to U)$ with side effects, such as:

- references, or
- exceptions

Equirecursive types in OCaml

OCaml has both isorecursive and equirecursive types.

- equirecursive types are restricted by default to objects or datatypes.
- unrestricted equirecursive types are available upon explicit request.

Quiz: why so?

Isorecursive types

The folding/unfolding is witnessed by an explicit coercion.

The uniqueness rule is often omitted

(hence, the equality relation is weaker)

Encoding isorecursive types with ADT

The recursive type $\mu\beta.\tau$ can be represented in System F by introducing a datatype with a unique constructor:

$$\text{type } G\,\vec{\alpha} = \Sigma(C : \forall\vec{\alpha}.\,[\beta \mapsto G\,\vec{\alpha}]\tau \to G\,\vec{\alpha}) \qquad \text{where } \vec{\alpha} = \mathrm{ftv}(\tau) \setminus \{\beta\}$$

For any $\vec{\alpha}$, the constructor $C\vec{\alpha}$ coerces $[\beta \mapsto G\,\vec{\alpha}]\tau$ to $G\,\vec{\alpha}$ and the reverse coercion is the function $\lambda x{:}G\,\vec{\alpha}.\,d_G\,\vec{\alpha}\;x\;(\lambda y.\,y)$.

Since this datatype has a unique constructor, pattern matching always succeeds and amounts to the identity. Hence, in $\lceil F \rceil$, *the constructor could be removed: coercions have no computational content.*

Records

A record can be defined as

$$\text{type } G\,\vec{\alpha} = \Pi_{i\in 1..n}(\ell_i : \tau_i) \qquad\qquad \text{where } \vec{\alpha} = \bigcup_{i\in 1..n} \mathrm{ftv}(\tau_i)$$

Exercise

What are the corresponding declarations in System F?

- *a new type constructor* G_Π,
- 1 *constructor* $C_\Pi \;:\; \forall\vec{\alpha}.\, \tau_1 \to \ldots \tau_n \to G\,\vec{\alpha}$
- n *destructors* $d_{\ell_i} \;:\; \forall\vec{\alpha}.\, G\,\vec{\alpha} \to \tau_i$
- n *reduction rules* $d_{\ell_i}\,\bar{\tau}\;(C_\Pi\;\bar{\tau}\;v_1\;\ldots v_n) \;\rightsquigarrow\; v_i$

Can a record also be used for defining recursive types?

Exercise

Show type soundness for records.

Deep pattern matching

In practice, one allows deep pattern matching and wildcards in patterns.

```
type nat = Z | S of nat
let rec equal n1 n2 = match n1, n2 with
  | Z, Z → true
  | S m1, S m2 → equal m1 m2
  | _ → false
```

Then, one should check for *exhaustiveness* of pattern matching.

Deep pattern matching can be compiled away into shallow patterns—or directly compiled to efficient code.

See [Le Fessant and Maranget, 2001; Maranget, 2007]

Exercise

Do the transofrmation manually for the function equal.

ADTs Regular

$$\text{type } G\ \vec{\alpha} = \Sigma_{i\in 1..n}(C_i : \forall\vec{\alpha}.\,\tau_i \to G\ \vec{\alpha})$$

If all occurrences of G in τ_i are $G\ \vec{\alpha}$ then, the ADT is *regular*.

Remark regular ADTs can be encoded in System-F. (More precisely, the church encodings of regular ADTs are typable in System-F.)

ADTs — Non Regular

Non-regular ADT's do not have this restriction:

```
type 'a seq =
  | Nil
  | Zero of ('a * 'a) seq
  | One of 'a * ('a * 'a) seq
```

They usually need *polymorphic* recursion to be manipulated.

Non regular ADT are heavily used by Okasaki [1999] for implementing purely functional data structures.

(They are also typically used with GADTs.)

Non-regular ADT can actually be encoded in F^ω.

Contents

Existential types *Examples*

A frozen application returning a value of type ($\approx$ a thunk)

$$\exists\alpha.(\alpha \to \tau) \times \alpha$$

Type of closures in the environment-passing variant:

$$[\![\tau_1 \to \tau_2]\!] \quad = \quad \exists\alpha.((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha$$

A possible encoding of objects:

$$
\begin{array}{ll}
= \quad \exists\rho. & \rho \text{ describes the state} \\
\quad\quad \mu\alpha. & \alpha \text{ is the concrete type of the closure} \\
\quad\quad\quad \Pi\,(& \text{a tuple...} \\
\quad\quad\quad\quad \{(\alpha \times \tau_1) \to \tau_1'; & \text{... that begins with a record...} \\
\quad\quad\quad\quad\quad \ldots & \\
\quad\quad\quad\quad\quad (\alpha \times \tau_n) \to \tau_n'\ \}\ ; & \text{... of method code pointers...} \\
\quad\quad\quad\quad \rho & \text{...and continues with the state} \\
\quad\quad\quad) & \quad \text{(a tuple of unknown length)}
\end{array}
$$

Existential types

Let's first look at the type-erasing interpretation, with an explicit notation for introducing and eliminating existential types.

Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

$$\textsc{Pack}\quad \frac{\Gamma \vdash M : [\alpha \mapsto \tau']\tau}{\Gamma \vdash \textit{pack}\ \tau', M\ \textit{as}\ \exists\alpha.\,\tau : \exists\alpha.\,\tau}$$

$$\textsc{Unpack}\quad \frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma, \alpha, x : \tau_1 \vdash M_2 : \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \textit{let}\ \alpha, x = \textit{unpack}\ M_1\ \textit{in}\ M_2 : \tau_2}$$

Anything wrong?The side condition $\alpha \# \tau_2$ is mandatory here to ensure well-formedness of the conclusion.

The side condition may also be written $\Gamma \vdash \tau_2$ which implies $\alpha \# \tau_2$, given that the well-formedness of the last premise implies $\alpha \notin \mathrm{dom}(\Gamma)$.

Note the *imperfect* duality between universals and existentials:

$$\textsc{TAbs}\quad \frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\,M : \forall\alpha.\,\tau} \qquad \textsc{TApp}\quad \frac{\Gamma \vdash M : \forall\alpha.\,\tau}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}$$

On existential elimination

It would be nice to have a simpler elimination form, perhaps like this:

$$\frac{\Gamma, \alpha \vdash M : \exists\alpha.\tau}{\Gamma, \alpha \vdash \textit{unpack}\ M : \tau}$$

Informally, this could mean that, if M has type τ for some *unknown* α, then it has type τ, where α is "fresh"...

Why is this broken?

We could immediately *universally* quantify over α, and conclude that $\Gamma \vdash \Lambda\alpha.\textit{unpack}\ M : \forall\alpha.\tau$. This is nonsense!

Replacing the premise $\Gamma, \alpha \vdash M : \exists\alpha.\tau$ by the conjunction $\Gamma \vdash M : \exists\alpha.\tau$ and $\alpha \in \mathrm{dom}(\Gamma)$ would make the rule even more permissive, so it wouldn't help.

On existential elimination

A correct elimination rule must force the existential package to be *used* in a way that does not rely on the value of α.

Hence, the elimination rule must have control over the *user* of the package—that is, over the term M_2.

$$\textsc{Unpack}\quad \frac{\begin{array}{c}\Gamma \vdash M_1 : \exists\alpha.\tau_1 \\ \Gamma, \alpha; x : \tau_1 \vdash M_2 : \tau_2 \qquad \alpha \# \tau_2\end{array}}{\Gamma \vdash \textit{let}\ \alpha, x = \textit{unpack}\ M_1\ \textit{in}\ M_2 : \tau_2}$$

The restriction $\alpha \# \tau_2$ prevents writing "*let* α, x = *unpack* M_1 *in* x", which would be equivalent to the unsound "*unpack* M" of the previous slide.

The fact that α is bound within M_2 forces it to be treated abstractly.

In fact, M_2 must be ??? in α.

On existential elimination

In fact, M_2 must be *polymorphic* in α: the second premise could be:

$$\frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma, \;\alpha, x : \tau_1 \;\vdash \Lambda\alpha.\,\lambda x{:}\tau_1.\,M_2 : \forall\alpha.\,\tau_1 \rightarrow \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \textit{let}\ \alpha, x = \textit{unpack}\ M_1\ \textit{in}\ M_2 : \tau_2}$$

or, if N_2 stands for $\Lambda\alpha.\,\lambda x{:}\tau_1.\,M_2$:

$$\frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma \vdash N_2 : \forall\alpha.\,\tau_1 \rightarrow \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \textit{unpack}\ M_1\ N_2 : \tau_2}$$

One could even view "$\textit{unpack}_{\exists\alpha.\tau_1}$" as a family of *constants* of types:

$$\textit{unpack}_{\exists\alpha.\tau_1} : \quad (\exists\alpha.\tau_1) \rightarrow \big(\forall\alpha.\,(\tau_1 \rightarrow \tau_2)\big) \rightarrow \tau_2 \qquad \alpha \# \tau_2$$

Thus, $\textit{unpack}_{\exists\alpha.\tau} : \quad \forall\beta.\,\big((\exists\alpha.\tau) \rightarrow (\forall\alpha.\,(\tau \rightarrow \beta)) \rightarrow \beta\big)$

or, better $\textit{unpack}_{\exists\alpha.\tau} : \quad (\exists\alpha.\tau) \rightarrow \forall\beta.\,\big((\forall\alpha.\,(\tau \rightarrow \beta)) \rightarrow \beta\big)$

β stands for τ_2: it is bound prior to α, so it cannot be instantiated to a type that refers to α, which reflects the side condition $\alpha \# \tau_2$.

On existential introduction

$$\frac{\Gamma \vdash M : [\alpha \mapsto \tau']\tau}{\Gamma \vdash \textit{pack}\ \tau', M\ \textit{as}\ \exists\alpha.\,\tau : \exists\alpha.\tau}\ \textsc{Pack}$$

Hence, "$\textit{pack}_{\exists\alpha.\tau}$" can be viewed as a family *constant* of types:

$$\textit{pack}_{\exists\alpha.\tau} : \quad [\alpha \mapsto \tau']\tau \to \exists\alpha.\tau$$

i.e. of polymorphic types:

$$\textit{pack}_{\exists\alpha.\tau} : \quad \forall\alpha.\,(\tau \to \exists\alpha.\tau)$$

Existentials as constants

In System F, existential types can be presented as a family of constants:

$$\begin{array}{rcl} \textit{pack}_{\exists\alpha.\tau} & : & \forall\alpha.(\tau \to \exists\alpha.\tau) \\ \textit{unpack}_{\exists\alpha.\tau} & : & \exists\alpha.\tau \to \forall\beta.((\forall\alpha.(\tau \to \beta)) \to \beta) \end{array}$$

Read:

- for *any* α, if you have a τ, then, for *some* α, you have a τ;
- if, for *some* α, you have a τ, then, (for any β,) if you wish to obtain a β out of it, you must present a function which, for *any* α, obtains a β out of a τ.

This is somewhat reminiscent of ordinary first-order logic: $\exists x.F$ is equivalent to, and can be defined as, $\neg(\forall x.\neg F)$.

Is there an encoding of existential types into universal types?

Encoding existentials into universals

The type translation is *double negation:*

$$[\![\exists\alpha.\tau]\!] = \forall\beta.\,((\forall\alpha.\,([\![\tau]\!] \to \beta)) \to \beta) \qquad \text{if } \beta \# \tau$$

The term translation is:

$$\begin{array}{rcl} [\![\mathit{pack}_{\exists\alpha.\tau}]\!] & : & \forall\alpha.\,([\![\tau]\!] \to [\![\exists\alpha.\tau]\!]) \\ & = & \Lambda\alpha.\,\lambda x{:}[\![\tau]\!].\,\Lambda\beta.\,\lambda k{:}\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\ \alpha\ x \\ [\![\mathit{unpack}_{\exists\alpha.\tau}]\!] & : & [\![\exists\alpha.\tau]\!] \to \forall\beta.\,((\forall\alpha.\,([\![\tau]\!] \to \beta)) \to \beta) \\ & = & \lambda x{:}[\![\exists\alpha.\tau]\!].\,x \end{array}$$

There is little choice, if the translation is to be type-preserving.

What is the computational content of this encoding?

A *continuation-passing transform.*

This encoding is due to Reynolds [1983],
although it has more ancient roots in logic.

The semantics of existential types as constants

$\mathit{pack}_{\exists\alpha.\tau}$ can be treated as a unary constructor, and $\mathit{unpack}_{\exists\alpha.\tau}$ as a unary destructor. The δ-reduction rule is:

$$\mathit{unpack}_{\exists\alpha.\tau_0}\,(\mathit{pack}_{\exists\alpha.\tau}\tau'\,V) \;\longrightarrow\; \Lambda\beta.\,\lambda y{:}\forall\alpha.\,\tau\to\beta.\,y\;\tau'\;V$$

It would be more intuitive, however, to treat $\mathit{unpack}_{\exists\alpha.\tau_0}$ as a binary destructor:

$$\mathit{unpack}_{\exists\alpha.\tau_0}\,(\mathit{pack}_{\exists\alpha.\tau}\tau'\,V)\;\tau_1\;(\Lambda\alpha.\,\lambda x{:}\tau.\,M) \;\longrightarrow\; [\alpha\mapsto\tau'][x\mapsto V]M$$

Remark:

- This does not quite fit in our generic framework for constants, which must receive all type arguments prior to value arguments.
- But our framework could be easily extended.

The semantics of existential types as primitive

We extend values and evaluation contexts as follows:

$$\begin{array}{lcl} V & ::= & \ldots \mid \mathit{pack}\,\tau', V\ \mathit{as}\,\tau \\ E & ::= & \ldots \mid \mathit{pack}\,\tau', [\,]\ \mathit{as}\,\tau \mid \mathit{let}\,\alpha, x = \mathit{unpack}\,[\,]\ \mathit{in}\,M \end{array}$$

We add the reduction rule:

$$\mathit{let}\,\alpha, x = \mathit{unpack}\,(\mathit{pack}\,\tau', V\ \mathit{as}\,\tau)\ \mathit{in}\,M \longrightarrow [\alpha \mapsto \tau'][x \mapsto V]M$$

Exercise

Show that subject reduction and progress hold.

The semantics of existential types beware!

The reduction rule for existentials destructs its arguments.

Hence, *let* α, x = *unpack* M_1 *in* M_2 cannot be reduced unless M_1 is itself a packed expression, which is indeed the case when M_1 is a value (or in head normal form).

This contrasts with *let* $x : \tau$ = M_1 *in* M_2 where M_1 need not be evaluated and may be an application (*e.g.* with call-by-name or strong reduction strategies).

The semantics of existential types beware!

Exercise

Find an example that illustrates why the reduction of
let α, x = *unpack* M_1 *in* M_2 *could be problematic when* M_1 *is not a value.*

Need a hint?

Use a conditional *Solution*

Let M_1 be if M then V_1 else V_2 where V_i is of the form *pack* τ_i, W_i *as* $\exists\alpha.\tau$ and the two witnesses τ_1 and τ_2 differ.

There is no common type for the unpacking of the two possible results V_1 and V_2. The choice between those two possible results must be made, by evaluating M_1, before unpacking.

Is pack too verbose?

Exercise

Recall the typing rule for pack:

$$\frac{\Gamma \vdash M : [\alpha \mapsto \tau']\tau}{\Gamma \vdash \textit{pack}\, \tau', M \textit{ as } \exists\alpha.\, \tau : \exists\alpha.\tau}$$

Isn't the witness type τ' annotation superfluous?

- The type τ_0 of M is fully determined by M. Given the type $\exists\alpha.\tau$ of the packed value, checking that τ_0 is of the form $[\alpha \mapsto \tau']\tau$ is the matching problem for second-order types, which is simple.
- However, the reduction rule need the witness type τ'. If it were not available, it would have to be computed during reduction. The reduction rule would then not be pure rewriting.

The explicitly-typed language need the witness type for simplicity, while in the surface language, it could be omitted and reconstructed.

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Implicitly-typed existential types

Intuitively, pack and unpack are just type annotations that could be dropped, leaving a let-binding instead of the unpack form.

Hence, the typing rule for implicitly-typed existential types:

$$\textsc{Unpack}\ \frac{\Gamma \vdash a_1 : \exists\alpha.\tau_1 \qquad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \mathit{let}\ x = a_1\ \mathit{in}\ a_2 : \tau_2} \qquad \textsc{Pack}\ \frac{\Gamma \vdash a : [\alpha \mapsto \tau']\tau}{\Gamma \vdash a : \exists\alpha.\tau}$$

Notice, however, that this let-binding is not typechecked as syntactic sugar for an immediate application!

The semantics of this let-binding is as before:

$$E ::= \ldots \mid \mathit{let}\ x = E\ \mathit{in}\ M \qquad \mathit{let}\ x = V\ \mathit{in}\ M \longrightarrow [x \mapsto V]M$$

Is the semantics type-erasing?

Implicitly-typed existential types

subtlety

Yes, it is.

But there is a subtlety! What about the call-by-name semantics?

We chose a call-by-value semantics, but so far, as long as there is no side-effect, we could have chosen a call-by-name semantics (or even perform reduction under abstraction).

In a call-by-name semantics, the let-bound expression is not reduced prior to substitution in the body:

$$\textit{let}\ x = M_1\ \textit{in}\ M_2 \longrightarrow [x \mapsto M_1]M_2$$

With existential types, this breaks subject reduction!

Why?

Implicitly-typed existential types subtlety

Let τ_0 be $\exists\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$ and v_0 a value of type *bool*. Let v_1 and v_2 be two values of type τ_0 with incompatible witness types, *e.g.* $\lambda f.\, \lambda x.\, 1 + (f\ (1 + x))$ and $\lambda f.\, \lambda x.\, \mathit{not}\ (f\ (\mathit{not}\ x))$.

Let v be the function $\lambda b.$ if b then v_1 else v_2 of type *bool* $\to \tau_0$.

$$a_1 \quad = \quad \textit{let } x = v\ v_0 \textit{ in } x\ (x\ (\lambda y.\, y)) \quad \longrightarrow \quad v\ v_0\ (v\ v_0\ (\lambda y.\, y)) \quad = \quad a_2$$

We have $\varnothing \vdash a_1 : \exists\alpha.\, \alpha \to \alpha$ while $\varnothing \nvdash a_2 : \tau$.

What happened? The term a_1 is well-typed since $v\ v_0$ has type τ_0, hence x can be assumed of type $(\beta \to \beta) \to (\beta \to \beta)$ for some unknown type β and $\lambda y.\, y$ is of type $\beta \to \beta$.

However, without the outer existential type $v\ v_0$ can only be typed with $(\forall\alpha.\, \alpha \to \alpha) \to \exists\alpha.\, (\alpha \to \alpha)$, because the value returned by the function need different witnesses for α. This is demanding too much on its argument and the outer application is ill-typed.

Implicitly-typed existential types subtlety

One could wonder whether the syntax should not allow the implicit introduction of unpacking (instead of requesting a let-binding).

One could argue that if some expression is the expansion of a well-typed let-binding, then it should also be well-typed:

$$\frac{\Gamma \vdash a_1 : \exists \alpha.\tau_1 \qquad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash [x \mapsto a_1]a_2 : \tau_2}$$

Comments?

- This rule does not have a logical flavor...
- It fixes the previous example, but not the general case:
  Pick a_1 *that is not yet a value after one reduction step.*
  Then, after let-expansion, reduce one of the two occurrences of a_1*.*
  The result is no longer of the form $[x \mapsto a_1]a_2$*.*

Implicitly-typed existential types

subtlety

Existential types are trickier than they may appear at first.

The subject reduction property breaks if reduction is not restricted to expressions in head-normal forms.

Unrestricted reduction is still safe because well-typedness may eventually be recovered by further reduction steps—so that progress will never breaks.

Implicitly-typed existential types encoding

Notice that the CPS encoding of existential types (1) enforces the evaluation of the packed value (2) before it can be unpacked (3) and substituted (4):

$$
\begin{array}{rcll}
[\![\textit{unpack}\, a_1\ (\lambda x.\, a_2)]\!] & = & [\![a_1]\!]\ (\lambda x.\, [\![a_2]\!]) & \mathbf{(1)} \\
& \longrightarrow & (\lambda k.\, [\![a]\!]\ k)\ (\lambda x.\, [\![a_2]\!]) & \mathbf{(2)} \\
& \longrightarrow & (\lambda x.\, [\![a_2]\!])\ [\![a]\!] & \mathbf{(3)} \\
& \longrightarrow & [x \mapsto [\![a]\!]][\![a_2]\!] & \mathbf{(4)}
\end{array}
$$

In the call-by-value setting, $\lambda k.\, [\![a]\!]\ k$ would come from the reduction of $[\![\textit{pack}\, a]\!]$, *i.e.* is $(\lambda k.\, \lambda x.\, k\ x)\ [\![a]\!]$, so that a is always a value v.

However, a need not be a value. What is essential is that a_1 be reduced to some head normal form $\lambda k.\, [\![a]\!]\ k$.

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Iso-existential types in ML

What if one wished to extend ML with existential types?

Full type inference for existential types is undecidable, just like type inference for universals.

However, introducing existential types in ML is easy if one is willing to rely on user-supplied *annotations* that indicate *where* and *how* to pack and unpack.

Iso-existential types in ML

This *iso-existential* approach was suggested by Läufer and Odersky [1994].

Iso-existential types are explicitly *declared:*

$$D\,\vec{\alpha} \approx \exists\bar{\beta}.\tau \qquad \text{if } \mathrm{ftv}(\tau) \subseteq \bar{\alpha} \cup \bar{\beta} \quad \text{and} \quad \bar{\alpha} \mathrel{\#} \bar{\beta}$$

This introduces two constants, with the following type schemes:

$$\begin{array}{rcl} \mathit{pack}_D & : & \forall\bar{\alpha}\bar{\beta}.\,\tau \to D\,\vec{\alpha} \\ \mathit{unpack}_D & : & \forall\bar{\alpha}\gamma.\,D\,\vec{\alpha} \to (\forall\bar{\beta}.\,(\tau \to \gamma)) \to \gamma \end{array}$$

(Compare with basic isorecursive types, where $\bar{\beta} = \varnothing$.)

Iso-existential types in ML

One point has been hidden on the previous slide. The "type scheme:"

$$\forall \bar{\alpha}\gamma.\, D\ \vec{\alpha} \to (\forall \bar{\beta}.\, (\tau \to \gamma)) \to \gamma$$

is in fact *not* an ML type scheme. How could we address this?

A solution is to make *unpack*$_D$ a (binary) primitive construct again (rather than a constant), with an *ad hoc* typing rule:

$$\textsc{Unpack}_D\quad \frac{\Gamma \vdash M_1 : D\ \vec{\tau} \qquad \Gamma \vdash M_2 : \forall \bar{\beta}.\, ([\vec{\alpha} \mapsto \vec{\tau}]\tau \to \tau_2) \qquad \bar{\beta} \# \vec{\tau}, \tau_2}{\Gamma \vdash \textit{unpack}_D\ M_1\ M_2 : \tau_2} \qquad \text{where } D\ \vec{\alpha} \approx \exists \bar{\beta}.\tau$$

We have seen a version of this rule in System F earlier; this is an ML(-like) version.

The term M_2 must be polymorphic, which $\textsc{Gen}$ can prove.

Iso-existential types in ML (type inference, skip)

Iso-existential types are perfectly compatible with ML type inference.

The constant pack_D admits an ML type scheme, so it is unproblematic.

The construct unpack_D leads to this constraint generation rule (see type inference):

$$\langle\!\langle \mathit{unpack}_D\ M_1\ M_2 : \tau_2 \rangle\!\rangle \quad = \quad \exists \bar{\alpha}. \left(\begin{array}{l} \langle\!\langle M_1 : D\ \vec{\alpha} \rangle\!\rangle \\ \forall \bar{\beta}. \langle\!\langle M_2 : \tau \to \tau_2 \rangle\!\rangle \end{array} \right)$$

where $D\ \vec{\alpha} \approx \exists \bar{\beta}.\tau$ and, *w.l.o.g.*, $\bar{\alpha}\bar{\beta} \# M_1, M_2, \tau_2$.

A universally quantified constraint appears where polymorphism is *required.*

Iso-existential types in ML

In practice, Läufer and Odersky suggest fusing iso-existential types with algebraic data types.

This can be done in OCaml using GADTs (see last part of the course). The syntax for this in OCaml is:

$$\textit{type}\ D\ \vec{\alpha} = \ell : \tau\ \to D\ \vec{\alpha}$$

where ℓ is a data constructor and $\bar{\beta}$ appears free in τ but does not appear in $\vec{\alpha}$. The elimination construct is typed as:

$$\langle\!\langle \textit{match}\ M_1\ \textit{with}\ \ell\ x \to M_2 : \tau_2 \rangle\!\rangle \quad = \quad \exists \bar{\alpha}. \left(\begin{array}{l} \langle\!\langle M_1 : D\ \vec{\alpha} \rangle\!\rangle \\ \forall \bar{\beta}. \textit{def}\ x : \tau\ \textit{in}\ \langle\!\langle M_2 : \tau_2 \rangle\!\rangle \end{array} \right)$$

where, w.l.o.g., $\bar{\alpha}\bar{\beta} \# M_1, M_2, \tau_2$.

An example

Define $\mathit{Any} \approx \exists\beta.\beta$. An attempt to extract the raw content of a package fails:

$$\begin{array}{rcl} \langle\!\langle \mathit{unpack}_{\mathit{Any}}\, M_1\ (\lambda x.\, x) : \tau_2 \rangle\!\rangle & = & \langle\!\langle M_1 : \mathit{Any} \rangle\!\rangle \wedge \forall\beta.\, \langle\!\langle \lambda x.\, x : \beta \to \tau_2 \rangle\!\rangle \\ & \Vdash & \forall\beta.\, \beta = \tau_2 \\ & \equiv & \mathit{false} \end{array}$$

(Recall that $\beta \# \tau_2$.)

An example

Define

$$D\ \alpha \approx \exists\beta.(\beta \to \alpha) \times \beta$$

A client that regards β as abstract succeeds:

$$\begin{array}{cl} & \langle\!\langle \textit{unpack}_D\ M_1\ (\lambda(f,y).\ f\ y) : \tau \rangle\!\rangle \\ = & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \forall\beta.\ \langle\!\langle \lambda(f,y).\ f\ y : ((\beta \to \alpha) \times \beta) \to \tau \rangle\!\rangle) \\ \equiv & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \forall\beta.\ \textit{def}\ f : \beta \to \alpha; y : \beta\ \textit{in}\ \langle\!\langle f\ y : \tau \rangle\!\rangle) \\ \equiv & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \forall\beta.\ \tau = \alpha) \\ \equiv & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \tau = \alpha) \\ \equiv & \langle\!\langle M_1 : D\ \tau \rangle\!\rangle \end{array}$$

Existential types calls for universal types!

Exercise Let $\mathit{thunk}\,\alpha \approx \exists\beta.(\beta \to \alpha) \times \beta$ be the type of frozen computations. Assume given a list l with elements of type $\mathit{thunk}\,\tau_1$.

Assume given a function g of type $\tau_1 \to \tau_2$. Transform the list l into a new list l' of frozen computations of type $\mathit{thunk}\,\tau_2$ (without actually running any computation).

```
List.map (λ(z) let Delay (f, y) = z in Delay ((λ(z) g (f z)), y))
```

Try generalizing this example to a function that receives g and l and returns l' : it does not typecheck...

```
let lift g l =
  List.map (λ(z) let Delay (f, y) = z in Delay ((λ(z) g (f z)), y))
```

In expression *let* α, x = *unpack* M_1 *in* M_2, occurrences of x in M_2 can only be passed to external functions (free variables) that are polymorphic in α so that α does not leak out of its context.

Limits of iso-encodings

Using datatypes for existential and especially universal types is a simple solution to make them compatible with ML, but it comes with some limitations:

- All types must be declared before being used
- Programs become quite verbose, with many constructors that amount to writting type annotations, but in a more rigid way
- In particular, there is no canonical way of representing them. For exemple, a thunk of type $\exists\beta(\beta \to \mathit{int}) \times \beta$ could have been defined as Delay (succ, 1) where Delay is either one of

  ```
  type int_thunk = Delay : ('b → int) ∗ 'b → int_thunk
  type 'a thunk = Delay : ('b → 'a) ∗ 'b → 'a thunk
  ```

  but the two types are incompatible.

Hence, other primitive solutions have been considered, especially for universal types.

Uses of existential types

Mitchell and Plotkin [1988] note that existential types offer a means of explaining *abstract types.* For instance, the type:

$$\begin{aligned}\exists stack.\{&empty : stack;\\ &push : int \times stack \to stack;\\ &pop : stack \to option\ (int \times stack)\}\end{aligned}$$

specifies an abstract implementation of integer stacks.

Unfortunately, it was soon noticed that the elimination rule is too awkward, and that existential types alone do not allow designing *module systems* [Harper and Pierce, 2005].

Montagu and Rémy [2009] make existential types *more flexible* in several important ways, and argue that they might explain modules after all.

Rossberg, Russo, and Dreyer show that after all, generative modules can be encoded into System F with existential types [Rossberg et al., 2014].

Existential types in OCaml

Existential types are available indirectly in OCaml as a degenerate case of GADT and via abstract types and first-class modules.

Via GADT (iso-existential types)

```
type 'a thunk = Delay : ('b → 'a) * 'b → 'a thunk
let freeze f x = Delay (f, x)
let unfreeze (Delay (f, x)) = f x
```

Via first-class modules (abstract types)

```
module type Thunk = sig type b type a val f : b → a val x : b end
let freeze (type u) (type v) f  x =
  (module struct type b =  u type a = v  let f = f let x = x end
   : Delay)
let unfreeze (type u) (module M : Thunk with type a = u) = M.f M.x
```

Contents

An introduction to GADTs

What are they?

ADTs

Types of constructors are surjective: all types can potentially be reached

$$\begin{array}{l}\textbf{type}\ \alpha\ \text{list} = \\ \quad \mid \text{Nil}\ :\ \alpha\ \text{list} \\ \quad \mid \text{Const} : \alpha * \alpha\ \text{list} \to \alpha\ \text{list}\end{array}$$

GADTs

This is no more the case with GADTs

$$\begin{array}{l}\textbf{type}\ (\alpha, \beta)\ \text{eq} = \\ \quad \mid \mathit{Eq} : (\alpha, \alpha)\ \text{eq} \\ \quad \mid \text{Any} : (\alpha, \beta)\ \text{eq}\end{array}$$

The *Eq* constructor may only build values of types of (α, α) eq.
For example, it cannot build values of type (int, string) eq.

The criteria is *per constructor*: it remains a GADT when another (even *regular*) constructor is added.

Examples Defunctionalization

```
let add (x, y) = x + y in
let not x = if x then false else true in
let body b =
  let step x =
    add (x, if not b then 1 else 2)
  in step (step 0))
in body true
```

Introduce a constructor per function

```
type (_, _) apply =
  | Fadd  : (int * int, int) apply
  | Fnot  : (bool, bool) apply
  | Fbody : (bool, int) apply
  | Fstep : bool → (int, int) apply
```

Define a single apply function that dispatches all function calls:

```
let rec apply : type a b. (a, b) apply → a → b = fun f arg →
  match f with
  | Fadd    → let x, y = arg in x + y
  | Fnot    → let x = arg in if x then false else true
  | Fstep b → let x = arg in
              apply Fadd (x, if apply Fnot b then 1 else 2)
  | Fbody   → let b = arg in
              apply (Fstep b) (apply (Fstep b) 0)
in apply Fbody true
```

Examples   Typed evaluator

A typed abstract-syntax tree

```
type _ expr =
  | Int     : int → int expr
  | Zerop   : int expr → bool expr
  | If      : (bool expr * α expr * α expr) → α expr
let e0 : int expr = (If (Zerop (Int 0), Int 1, Int 2))
```

A typed evaluator (with no failure)

```
let rec eval : type a . a expr → a = fun x → match x with
  | Int x          → x                                   (* a = int  *)
  | Zerop x        → eval x > 0                          (* a = bool *)
  | If (b, e1, e2) → if eval b then eval e1 else eval e2
let b0 = eval e0
```

Exercise

What would you have to do without GADTs? Define a typed abstract syntax tree for the simply-typed λ-calculus and a *typed* evaluator.

Examples Generic programming

Example of printing

```
type _ ty =
  | Tint : int ty
  | Tbool : bool ty
  | Tlist : α ty → (α list) ty
  | Tpair : α ty * β ty → (α * β) ty

let rec to_string : type a. a ty → a → string = fun t x → match t with
  | Tint → string_of_int x
  | Tbool → if x then "true" else "false"
  | Tlist t → "[" ^ String.concat "; " (List.map (to_string t) x) ^ "]"
  | Tpair (a, b) →
      let u, v = x in "(" ^ to_string a u ^ ", " ^ to_string b v ^ ")"

let s = to_string (Tpair (Tlist Tint, Tbool)) ([1; 2; 3], true)
```

Examples Encoding sum types

type (α, β) sum = Left of α | Right of β

can be encoded as a product:

type (_, _, _) tag = Ltag : (α, α, β) tag | Rtag : (β, α, β) tag
type (α, β) prod = Prod : (γ, α, β) tag $* \gamma \rightarrow (\alpha, \beta)$ prod

let sum_of_prod (**type** a b) (p : (a, b) prod) : (a, b) sum =
  let Prod (t, v) = p **in** match t with Ltag → Left v | Rtag → Right v

Prod is a single, hence superfluous constructor: it need not be allocated.

A field common to both cases can be accessed without looking at the tag!

type (α, β) prod = Prod : (γ, α, β) tag $* \gamma *$ bool $\rightarrow (\alpha, \beta)$ prod
let get (**type** a b) (p : (a, b) prod) : bool =
  let Prod (t, v, s) = p **in** s

Examples Encoding sum types

Exercise

Specialize the encoding of sum types to the encoding of 'a list

Other uses of GADTs

GADTs

- May encode data-structure invariants, such as the state of an automaton, as illustrated by Pottier and Régis-Gianas [2006] for typechecking LR-parsers.
- They may be used to implement a form of dynamic type (similarly to the generic printer)
- They may be used to optimize representation (*e.g.* sum's encoding)
- GADTs can be used to encode type classes, using a technique analogous to defunctionalization [Pottier and Gauthier, 2006].

Reducing GADTs to type equality (and existential types)

All GADTs can be encoded with a single one, encoding type equality:

```
type (α, β) eq = Eq : (α, α) eq
```

For instance, generic programming can then be redefined as follows:

```
type α ty =
  | Tint  : (α, int) eq → α ty                              (* int ty            *)
  | Tlist : (α, β list) eq * β ty → α ty                    (* α ty → α list ty *)
  | Tpair : (α, (β * γ)) eq * β ty * γ ty → α ty
```

This declaration is not a GADT, just an existential type!

- ▷ We enlarge the domain of each constructor,
- ▷ But require a proof evidence as an extra argument that a certain equality holds *to restrict the possible uses* of the constructors.

```
let rec to_string : type a. a ty → a → string = fun t x → match t with
  | Tint Eq          → string_of_int x
  | Tlist (Eq, l)    → "[" ^String.concat "; " (List.map (to_string l) x)^ "]"
  | Tpair (Eq,a,b) →
      let u, v = x in "(" ^ to_string a u ^ ", " ^ to_string b v ^ ")"

let s = to_string (Tpair (Eq, Tlist (Eq, Tint Eq), Tint Eq)) ([1; 2; 3], 0)
```

Reducing GADTs to type equality (and existential types)

All GADTs can be encoded with a single one :

type (α, β) eq $=$ *Eq* : (α, α) eq

For instance, generic programming can be redefined as follows:

type α ty $=$
| Tint : $(\alpha$, int$)$ eq $\rightarrow \alpha$ ty
| Tlist : $(\alpha, \beta$ list$)$ eq $* \beta$ ty $\rightarrow \alpha$ ty
| Tpair : $(\alpha, (\beta * \gamma))$ eq $* \beta$ ty $* \gamma$ ty $\rightarrow \alpha$ ty

This declaration is not a GADT, just an existential type!

let rec to_string : **type** a. a ty → a → string = **fun** t x → match t with
| Tint *Eq* → string_of_int x
| Tlist (*Eq*, l) → ...
| Tpair (*Eq*, a, b) → ...

▷ Pattern “Tint *Eq*” is GADT matching

Reducing GADTs to type equality (and existential types)

All GADTs can be encoded with a single one :

type (α, β) eq = *Eq* : (α, α) eq

For instance, generic programming can be redefined as follows:

type α ty =
| Tint : $(\alpha$, int) eq → α ty
| Tlist : $(\alpha, \beta$ list) eq ∗ β ty → α ty
| Tpair : $(\alpha, (\beta * \gamma))$ eq ∗ β ty ∗ γ ty → α ty

This declaration is not a GADT, just an existential type!

let rec to_string : **type** a. a ty → a → string = **fun** t x → match t with
| Tint p → **let** *Eq* = p **in** string_of_int x
| Tlist (*Eq*, l) → ...
| Tpair (*Eq*, a, b) → ...

- ▷ Pattern "Tint *Eq*" is GADT matching
- ▷ **let** *Eq* = p **in**.. introduces the equality a = int in the current branch

Formalisation of GADTs

We can extend System F with type equalities to encode GADTs.

We *cannot* encode type equalities in System F—but in System F^ω: they bring something more, namely *local equalities* in the typing context.

We write $\tau_1 \sim \tau_2$ for (τ_1, τ_2) eq

When typechecking an expression

$$E[\mathit{let}\ x : \tau_1 \sim \tau_2 = M_0\ \mathit{in}\ M] \qquad\qquad E[\lambda x : \tau_1 \sim \tau_2 .\, M]$$

- M is typechecked with the asumption that $\tau_1 \sim \tau_2$, *i.e.* types τ_1 and τ_2 are equivalent, which allows for type conversion within M
- but E and M_0 are typechecked without this asumption
- What is learned by an equation remains local to its static scope, and does not extend to its surrounding context (or the rest of the program execution trace).

Fc (simplified) *Add equality coercions to System F*

Types

$$\tau ::= \ldots \mid \tau_1 \sim \tau_2$$

Expressions

$$M ::= \ldots \mid \gamma \triangleleft M \mid \gamma$$

Coercions are first-class and can be applied to terms.

Coercions witness type equivalences:

$\gamma ::= \alpha$	variable
$\mid \langle\tau\rangle$	reflexivity
$\mid \mathsf{sym}\,\gamma$	symmetry
$\mid \gamma_1 ; \gamma_2$	transitivity
$\mid \gamma_1 \to \gamma_2$	arrow coercions
$\mid \mathsf{left}\,\gamma$	left projection
$\mid \mathsf{right}\,\gamma$	right projection
$\mid \forall\alpha.\gamma$	type generalization
$\mid \gamma @ \tau$	type instantiation

Typing rules:

$$\textsc{Coerce}\quad \frac{\Gamma \vdash M : \tau_1 \qquad \Gamma \vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash \gamma \triangleleft M : \tau_2}$$

$$\textsc{Coercion}\quad \frac{\Gamma \Vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash \gamma : \tau_1 \sim \tau_2}$$

$$\textsc{CoAbs}\quad \frac{\Gamma, x : \tau_1 \sim \tau_2 \vdash M : \tau}{\Gamma \vdash \lambda x : \tau_1 \sim \tau_2 . M : \tau_1 \sim \tau_2 \to \tau}$$

Fc (simplified) — Typing of coercions

Eq-Hyp
$$\frac{y : \tau_1 \sim \tau_2 \in \Gamma}{\Gamma \Vdash y : \tau_1 \sim \tau_2}$$

Eq-Ref
$$\frac{\Gamma \vdash \tau}{\Gamma \Vdash \langle\tau\rangle : \tau \sim \tau}$$

Eq-Sym
$$\frac{\Gamma \Vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \Vdash \mathsf{sym}\,\gamma : \tau_2 \sim \tau_1}$$

Eq-Trans
$$\frac{\Gamma \Vdash \gamma_1 : \tau_1 \sim \tau \qquad \Gamma \Vdash \gamma_2 : \tau \sim \tau_2}{\Gamma \Vdash \gamma_1 ; \gamma_2 : \tau_1 \sim \tau_2}$$

Eq-Arrow
$$\frac{\Gamma \Vdash \gamma_1 : \tau_1' \sim \tau_1 \qquad \Gamma \Vdash \gamma_2 : \tau_2 \sim \tau_2'}{\Gamma \Vdash \gamma_1 \to \gamma_2 : \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

Eq-Left
$$\frac{\Gamma \Vdash \gamma : \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}{\Gamma \Vdash \mathsf{left}\,\gamma : \tau_1' \sim \tau_1}$$

Eq-Right
$$\frac{\Gamma \Vdash \gamma : \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}{\Gamma \Vdash \mathsf{right}\,\gamma : \tau_2 \sim \tau_2'}$$

Eq-All
$$\frac{\Gamma, \alpha \Vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \Vdash \forall\alpha.\,\gamma : \forall\alpha.\,\tau_1 \sim \forall\alpha.\,\tau_2}$$

Eq-Inst
$$\frac{\Gamma \Vdash \gamma : \forall\alpha.\,\tau_1 \sim \forall\alpha.\,\tau_2 \qquad \Gamma \vdash \tau}{\Gamma \Vdash \gamma @ \tau : [\alpha \mapsto \tau]\tau_1 \sim [\alpha \mapsto \tau]\tau_2}$$

Only equalities between *injective* type constructors can be decomposed.

Semantics

Coercions should be without computational content

- they are just type information, and should be erased at runtime
- they should not block redexes
- in Fc, we may always push them down inside terms, adding new reduction rules:

$$
\begin{array}{rcl}
(\gamma \triangleleft V_1)\; V_2 & \longrightarrow & \mathsf{right}\,\gamma \triangleleft (V_1\; (\mathsf{left}\,\gamma \triangleleft V_2)) \\
(\gamma \triangleleft V)\; \tau & \longrightarrow & (\gamma @ \tau) \triangleleft (V\; \tau) \\
\gamma_1 \triangleleft (\gamma_2 \triangleleft V) & \longrightarrow & (\gamma_1 ; \gamma_2) \triangleleft V
\end{array}
$$

Semantics

Coercions should be without computational content

Except for coercion abstractions that must stop the evaluation

- Otherwise, one could attempt to reduce M in $\lambda \mathit{int} \sim \mathit{bool}.\, M$ when M is *not* $(\mathit{bool} \triangleleft 0)$, which is well-typed in this context.
- In call-by-value,

$$\begin{array}{lll} \lambda x : \tau_1 \sim \tau_2 .\, M & \text{freezes} & \text{the evaluation of } M, \\ M \triangleleft \gamma & \text{resumes} & \text{the evaluation of } M. \end{array}$$

  Must always be enforced, even with other strategies
- Full reduction *at compile time* may still be perfomed, but be aware of stuck programs and treat them as dead branches.

Type soundness *Syntactic proofs*

Type soundness

By subject reduction and progress with explicit coercions

Erasing semantics

Important and not so obvious.

$$\begin{array}{lll} \gamma \triangleleft M & \text{erases} & \text{to } M \\ \gamma & \text{erases} & \text{to } \diamond \end{array}$$

Slogan that "coercion have 0-bit information", *i.e.*
Coercions need not be passed at runtime—-but still block the reduction.
Expressions and typing rules.

$$\text{Coerce}\ \frac{\Gamma \vdash M : \tau_1 \qquad \Gamma \vdash \diamond : \tau_1 \sim \tau_2}{\Gamma \vdash M : \tau_2} \qquad \text{Coercion}\ \frac{\Gamma \Vdash \tau_1 \sim \tau_2}{\Gamma \vdash \diamond : \tau_1 \sim \tau_2} \qquad \text{CoAbs}\ \frac{\Gamma, x : \tau_1 \sim \tau_2 \vdash M : \tau}{\Gamma \vdash \lambda x : \tau_1 \sim \tau_2 .\, M : \tau_1 \sim \tau_2 \to \tau}$$

Type soundness *Syntactic proofs*

The introduction of type equality constraints in System F has been introduced and formalized by Sulzmann et al. [2007].

Scherer and Rémy [2015] show how strong reduction and confluence can be recovered in the presence of possibly uninhabited coercions.

Type soundness *Semantic proofs*

Equality coercions are a small logic of type conversions.

Type conversions may be enriched with more operations.

A very general form of coercions has been introduced by Cretin and Rémy [2014].

The type soundness proof became too cumbersome to be conducted syntactically.

Instead a semantic proof is used, interpreting types as sets of terms (a technique similar to unary logical relations)

Type checking / inference

With explicit coercions, types are fully determined from expressions.

However, the user prefers to leave applications of COERCE implicit.

Then types becomes ambiguous: when leaving the scope of an equation: which form should be used, among the equivalent ones?

This must be determined from the context, including the return type, and calls for extra type annotations:

```
let rec eval : type a . a expr → a = fun x → match x with
  | Int x          → x    (* x : int, but a = int, should we return x : a? *)
  | Zerop x        → eval x > 0
  | If (b, e1, e2) → if eval b then eval e1 else eval e2
```

In ML, type annotations must be used to tell

- the type of the context
- which datatypes must be typed as GADTs.

In Coq, one must use return type annotations on matches.

Type inference in ML-like languages with GADTs

Simonet and Pottier [2007] gave a presentation of type inference for GADTs with general typing constraints for ML-like languages.

Pottier and Régis-Gianas [2006] introduced a stratified approach to better propagate constraints from outisde to inside GADTs contexts.

Vytiniotis et al. [2011] introduced the outside-in approach, used in Haskell, which restricts type information to flow from outside to inside GADT contexts.

Garrigue and Rémy [2013] introduced the notion of ambivalent types, used in OCaml, to restrict type occurrences that must be considered ambiguous and explicitly specified using type annotations.

Contents

Type-preserving compilation

Compilation is type-preserving when each intermediate language is *explicitly typed*, and each compilation phase transforms a typed program into a typed program in the next intermediate language.

Why *preserve types* during compilation?

- it can help debug the compiler;
- types can be used to drive optimizations;
- types can be used to produce *proof-carrying code;*
- proving that types are preserved can be the first step towards proving that the *semantics* is preserved [Chlipala, 2007].

Type-preserving compilation

Type-preserving compilation exhibits an encoding of programming constructs into programming languages with usually richer type systems.

The encoding may sometimes be used directly as a programming idiom in the source language.

For example:

- Closure conversion requires an extension of the language with existential types, which happens to be very useful on their own.
- Closures are themselves a simple form of objects, which can also be explained with existential types.
- Defunctionalization may be done manually on some particular programs, *e.g.* in web applications to monitor the computation.

Type-preserving compilation

A classic paper by Morrisett *et al.* [1999] shows how to go from System F to Typed Assembly Language, while preserving types along the way. Its main passes are:

- *CPS conversion* fixes the order of evaluation, names intermediate computations, and makes all function calls tail calls;
- *closure conversion* makes environments and closures explicit, and produces a program where all functions are closed;
- allocation and initialization of tuples is made explicit;
- the calling convention is made explicit, and variables are replaced with (an unbounded number of) machine registers.

Translating types

In general, a type-preserving compilation phase involves not only a translation of *terms,* mapping M to $[\![M]\!]$, but also a translation of *types,* mapping τ to $[\![\tau]\!]$, with the property:

$$\Gamma \vdash M : \tau \quad \text{implies} \quad [\![\Gamma]\!] \vdash [\![M]\!] : [\![\tau]\!]$$

The translation of types carries a lot of information: examining it is often enough to guess what the translation of terms will be.

See the old lecture on type closure conversion.

Closure conversion

First-class functions may appear in the body of other functions. hence, their own body may contain free variables that will be bound to values during the evaluation in the execution environment.

Because they can be returned as values, and thus used outside of their definition environment, they must store their execution environment in their value.

A *closure* is the packaging of the code of a first-class function with its runtime environment, so that it becomes closed, *i.e.* independent of the runtime environment and can be moved and applied in another runtime environment.

Closures can also be used to represent recursive functions and objects (in the object-as-record-of-methods paradigm).

Source and target

In the following,

- the *source* calculus has *unary* λ-abstractions, which can have free variables;
- the *target* calculus has *binary* λ-abstractions, which must be *closed*.

Closure conversion can be easily extended to n-ary functions, or n-ary functions may be *uncurried* in a separate, type-preserving compilation pass.

Variants of closure conversion

There are at least two variants of closure conversion:

- in the *closure-passing variant*,
  the closure and the environment are a single memory block;
- in the *environment-passing variant*,
  the environment is a separate block, to which the closure points.

The impact of this choice on the translation of terms is minor.

Its impact on the translation of types is more important:
the closure-passing variant requires more type-theoretic machinery.

Closure-passing closure conversion

Let $\{x_1, \ldots, x_n\}$ be $\mathrm{fv}(\lambda x.\, a)$:

$$
\begin{array}{rcl}
[\![\lambda x.\, a]\!] & = & \textit{let } code = \lambda(clo, x). \\
 & & \quad \textit{let } (\_, x_1, \ldots, x_n) = clo \textit{ in } [\![a]\!] \textit{ in} \\
 & & (code, x_1, \ldots, x_n) \\[1em]
[\![a_1\ a_2]\!] & = & \textit{let } clo = [\![a_1]\!] \textit{ in} \\
 & & \textit{let } code = \textit{proj}_0\ clo \textit{ in} \\
 & & code\ (clo, [\![a_2]\!])
\end{array}
$$

(The variables $code$ and clo must be suitably fresh.)

Important! The layout of the environment must be known only at the closure allocation site, not at the call site. In particular, $\textit{proj}_0\ clo$ need not know the size of clo.

Environment-passing closure conversion

Let $\{x_1, \ldots, x_n\}$ be $\mathrm{fv}(\lambda x.\, a)$:

$$
\begin{array}{rcl}
[\![\lambda x.\, a]\!] & = & \textit{let } \mathit{code} = \lambda(\mathit{env}, x). \\
& & \quad \textit{let } \; (x_1, \ldots, x_n) = \mathit{env} \textit{ in } [\![a]\!] \textit{ in} \\
& & (\mathit{code}, (x_1, \ldots, x_n)) \\
[\![a_1 \; a_2]\!] & = & \textit{let } (\mathit{code}, \mathit{env}) = [\![a_1]\!] \textit{ in} \\
& & \mathit{code} \; (\mathit{env}, [\![a_2]\!])
\end{array}
$$

Questions: How can closure conversion be made *type-preserving?*

The key issue is to find a sensible definition of the type translation. In particular, what is the translation of a function type, $[\![\tau_1 \to \tau_2]\!]$?

Environment-passing closure conversion

Let $\{x_1, \ldots, x_n\}$ be $\text{fv}(\lambda x.\, a)$:

$$\begin{array}{rcl}\llbracket \lambda x.\, a \rrbracket & = & \textit{let } code = \lambda(env, x). \\ & & \quad \textit{let } \ (x_1, \ldots, x_n) = env \textit{ in } \llbracket a \rrbracket \textit{ in} \\ & & (code, (x_1, \ldots, x_n))\end{array}$$

Assume $\Gamma \vdash \lambda x.\, a : \tau_1 \to \tau_2$.
Assume, *w.l.o.g.*. $\text{dom}(\Gamma) = \text{fv}(\lambda x.\, a) = \{x_1, \ldots, x_n\}$.

Write $\llbracket \Gamma \rrbracket$ for the tuple type $x_1 : \llbracket \tau'_1 \rrbracket; \ldots; x_n : \llbracket \tau'_n \rrbracket$ where Γ is $x_1 : \tau'_1; \ldots; x_n : \tau'_n$. We also use $\llbracket \Gamma \rrbracket$ as a type to mean $\llbracket \tau'_1 \rrbracket \times \ldots \times \llbracket \tau'_n \rrbracket$.

We have $\Gamma, x : \tau_1 \vdash a : \tau_2$, so in environment $\llbracket \Gamma \rrbracket, x : \llbracket \tau_1 \rrbracket$, we have

- env has type $\llbracket \Gamma \rrbracket$,
- $code$ has type $(\llbracket \Gamma \rrbracket \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket$, and
- the entire closure has type $((\llbracket \Gamma \rrbracket \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket) \times \llbracket \Gamma \rrbracket$.

Now, *what should be the definition of $\llbracket \tau_1 \to \tau_2 \rrbracket$?*

Towards a type translation

Can we adopt this as a definition?

$$[\![\tau_1 \to \tau_2]\!] \quad = \quad (([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times [\![\Gamma]\!]$$

Naturally not. This definition is mathematically ill-formed: we cannot use Γ out of the blue.

That is, this definition is not uniform: it depends on Γ, *i.e.* the size and layout of the environment.

Do we really need to have a uniform translation of types?

Towards a type translation

Yes, we do.

We need a uniform translation of types, not just because it is nice to have one, but because it describes a *uniform calling convention.*

If closures with distinct environment sizes or layouts receive distinct types, then we will be unable to translate this well-typed code:

$$\textit{if} \ldots \textit{then}\ \lambda x.\, x + y\ \textit{else}\ \lambda x.\, x$$

Furthermore, we want function invocations to be translated uniformly, without knowledge of the size and layout of the closure's environment.

So, *what could be the definition of* $[\![\tau_1 \rightarrow \tau_2]\!]$*?*

The type translation

The only sensible solution is:

$$[\![\tau_1 \rightarrow \tau_2]\!] \quad = \quad \exists\alpha.((\alpha \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!]) \times \alpha$$

An *existential quantification* over the type of the environment abstracts away the differences in size and layout.

Enough information is retained to ensure that the application of the code to the environment is valid: this is expressed by letting the variable α occur twice on the right-hand side.

The type translation

The existential quantification also provides a form of *security*: the caller cannot do anything with the environment except pass it as an argument to the code; in particular, it cannot inspect or modify the environment.

For instance, in the source language, the following coding style guarantees that x remains even, no matter how f is used:

$$\textit{let } f = \textit{let } x = \textit{ref } 0 \textit{ in } \lambda().\, x := (x+2); !\, x$$

After closure conversion, the reference x is reachable via the closure of f. A malicious, untyped client could write an odd value to x. However, a *well-typed* client is unable to do so.

This encoding is not just type-preserving, but also *fully abstract:* it preserves (a typed version of) observational equivalence [Ahmed and Blume, 2008].

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Typed closure conversion

Everything is now set up to prove that, in System F with existential types:

$$\Gamma \vdash M : \tau \quad \text{implies} \quad [\![\Gamma]\!] \vdash [\![M]\!] : [\![\tau]\!]$$

Environment-passing closure conversion

Assume $\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2$ and $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\lambda x.\, M)$.

$$
\begin{array}{lcl}
[\![\lambda x{:}\tau_1.\, M]\!] & = & \textit{let}\ code : ([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
& & \quad \lambda(env : [\![\Gamma]\!], x : [\![\tau_1]\!]). \\
& & \qquad \textit{let}\ (x_1, \ldots, x_n : [\![\Gamma]\!]) = env\ \textit{in} \\
& & \qquad [\![M]\!] \\
& & \textit{in} \\
& & \textit{pack}\ [\![\Gamma]\!], (code, (x_1, \ldots, x_n)) \\
& & \ \textit{as}\ \exists\alpha.((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha
\end{array}
$$

We find $[\![\Gamma]\!] \vdash [\![\lambda x{:}\tau_1.\, M]\!] : [\![\tau_1 \to \tau_2]\!]$, as desired.

Environment-passing closure conversion

Assume $\Gamma \vdash M : \tau_1 \to \tau_2$ and $\Gamma \vdash M_1 : \tau_1$.

$$\begin{aligned}
[\![M\ M_1]\!] \quad = \quad & \textit{let}\ \alpha, (\mathit{code} : (\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!], \mathit{env} : \alpha) = \\
& \quad \textit{unpack}\ [\![M]\!]\ \textit{in} \\
& \mathit{code}\ (\mathit{env}, [\![M_1]\!])
\end{aligned}$$

We find $[\![\Gamma]\!] \vdash [\![M\ M_1]\!] : [\![\tau_2]\!]$, as desired.

Environment-passing closure conversion recursion

Recursive functions can be translated in this way, known as the "fix-code" variant [Morrisett and Harper, 1998] (leaving out type information):

$$\begin{array}{rcl} [\![\mu f.\lambda x.M]\!] & = & \textit{let rec } code\ (env, x) = \\ & & \quad \textit{let } f = \textit{pack}\ (code, env)\ \textit{in} \\ & & \quad \textit{let}\ (x_1, \ldots, x_n) = env\ \textit{in} \\ & & \quad [\![M]\!]\ \textit{in} \\ & & \textit{pack}\ (code, (x_1, \ldots, x_n)) \end{array}$$

where $\{x_1, \ldots, x_n\} = \mathrm{fv}(\mu f.\lambda x.M)$.

The translation of applications is unchanged: recursive and non-recursive functions have an identical calling convention.

What is the weak point of this variant?

A new closure is allocated at every call.

Environment-passing closure conversion recursion

Instead, the "fix-pack" variant [Morrisett and Harper, 1998] uses an extra field in the environment to store a back pointer to the closure:

$$
\begin{array}{rcl}
[\![\mu f.\lambda x.M]\!] & = & \textit{let}\ code\ (env, x) = \\
& & \quad \textit{let}\ (f, x_1, \ldots, x_n) = env\ \textit{in} \\
& & \quad [\![M]\!] \\
& & \textit{in} \\
& & \textit{let rec}\ clo = (code, (clo, x_1, \ldots, x_n))\ \textit{in} \\
& & clo
\end{array}
$$

where $\{x_1, \ldots, x_n\} = \mathrm{fv}(\mu f.\lambda x.M)$.

This requires general, recursively-defined *values.* Closures are now *cyclic* data structures.

Environment-passing closure conversion recursion

Here is how the "fix-pack" variant is type-checked. Assume $\Gamma \vdash \mu f.\lambda x.M : \tau_1 \to \tau_2$ and $\mathrm{dom}(\Gamma) = \{x_1, \dots, x_n\} = \mathrm{fv}(\mu f.\lambda x.M)$.

$$
\begin{array}{l}
[\![\mu f : \tau_1 \to \tau_2.\lambda x.M]\!] = \\
\qquad \textit{let code} : ([\![f : \tau_1 \to \tau_2; \Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
\qquad\quad \lambda(env : [\![f : \tau_1 \to \tau_2, \Gamma]\!], x : [\![\tau_1]\!]). \\
\qquad\qquad \textit{let } (f, x_1, \dots, x_n) : [\![f : \tau_1 \to \tau_2, \Gamma]\!] = env \textit{ in} \\
\qquad\qquad [\![M]\!] \textit{ in} \\
\qquad \textit{let rec clo} : [\![\tau_1 \to \tau_2]\!] = \\
\qquad\quad \textit{pack } [\![f : \tau_1 \to \tau_2, \Gamma]\!], (code, (clo, x_1, \dots, x_n)) \\
\qquad\quad \textit{as } \exists\alpha((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha) \\
\qquad \textit{in clo}
\end{array}
$$

Problem?

Environment-passing closure conversion recursion

The recursive function may be polymorphic, but recursive calls are monomorphic...

We can generalize the encoding afterwards,

$$[\![\Lambda\vec{\beta}.\,\mu f : \tau_1 \to \tau_2.\lambda x.M]\!] = \Lambda\vec{\beta}.\,[\![\mu f : \tau_1 \to \tau_2.\lambda x.M]\!]$$

whenever the right-hand side is well-defined.

This allows the *indirect* compilation of polymorphic recursive functions as long as the recursion is monomorphic.

Fortunately, the encoding can be straightforwardly adapted to *directly* compile polymorphically recursive functions into polymorphic closure.

Environment-passing closure conversion recursion

$$
\begin{array}{l}
[\![\mu f : \forall \vec{\beta}.\, \tau_1 \to \tau_2.\, \lambda x.M]\!] = \\
\quad \textit{let } code : \forall \vec{\beta}.\, ([\![f : \forall \vec{\beta}.\, \tau_1 \to \tau_2 ; \Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
\quad\quad \lambda(env : [\![f : \forall \vec{\beta}.\, \tau_1 \to \tau_2, \Gamma]\!], x : [\![\tau_1]\!]). \\
\quad\quad\quad \textit{let } (f, x_1, \ldots, x_n) : [\![f : \forall \vec{\beta}.\, \tau_1 \to \tau_2, \Gamma]\!] = env \textit{ in} \\
\quad\quad\quad [\![M]\!] \textit{ in} \\
\quad \textit{let rec } clo : [\![\forall \vec{\beta}.\, \tau_1 \to \tau_2]\!] = \\
\quad\quad \Lambda \vec{\beta}.\, \textit{pack } [\![f : \forall \vec{\beta}.\, \tau_1 \to \tau_2, \Gamma]\!], (code\ \vec{\beta}, (clo, x_1, \ldots, x_n)) \\
\quad\quad\quad \textit{as } \exists \alpha ((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha) \\
\quad \textit{in } clo
\end{array}
$$

The encoding is simple.

However, this requires the introduction of recursive non-functional values "let rec x = v". While this is a useful construct, it really alters the operational semantics and requires updating the type soundness proof.

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Closure-passing closure conversion

$$
\begin{array}{rcl}
[\![\lambda x.\, M]\!] & = & \textit{let } code = \lambda(clo, x). \\
& & \quad \textit{let } (\_, x_1, \ldots, x_n) = clo \textit{ in} \\
& & \quad [\![M]\!] \\
& & \textit{in } (code, x_1, \ldots, x_n) \\
\\
[\![M_1\ M_2]\!] & = & \textit{let } clo = [\![M_1]\!] \textit{ in} \\
& & \textit{let } code = \textit{proj}_0\ clo \textit{ in} \\
& & code\ (clo, [\![M_2]\!])
\end{array}
$$

There are two difficulties:

- a closure is a tuple, whose *first* field should be *exposed* (it is the code pointer), while the number and types of the remaining fields should be abstract;
- the first field of the closure contains a function that expects *the closure itself* as its first argument.

Closure-passing closure conversion

There are two difficulties:

- a closure is a tuple, whose *first* field should be *exposed* (it is the code pointer), while the number and types of the remaining fields should be abstract;
- the first field of the closure contains a function that expects *the closure itself* as its first argument.

What type-theoretic mechanisms could we use to describe this?

- existential quantification over the *tail* of a tuple (a.k.a. a *row*);
- *recursive types.*

Tuples, rows, row variables

The standard tuple types that we have used so far are:

$$
\begin{array}{rcll}
\tau & ::= & \ldots \mid \Pi\, R & \text{– types} \\
R & ::= & \epsilon \mid (\tau; R) & \text{– rows}
\end{array}
$$

The notation $(\tau_1 \times \ldots \times \tau_n)$ was sugar for $\Pi\,(\tau_1; \ldots; \tau_n; \epsilon)$.

Let us now introduce *row variables* and allow *quantification* over them:

$$
\begin{array}{rcll}
\tau & ::= & \ldots \mid \Pi\, R \mid \forall \rho. \tau \mid \exists \rho. \tau & \text{– types} \\
R & ::= & \rho \mid \epsilon \mid (\tau; R) & \text{– rows}
\end{array}
$$

This allows reasoning about the first few fields of a tuple whose length is not known.

Typing rules for tuples

The typing rules for tuple construction and deconstruction are:

$$\textsc{Tuple}\ \frac{\forall i.\ \in [1,n] \quad \Gamma \vdash M_i : \tau_i}{\Gamma \vdash (M_1, \ldots, M_n) : \Pi\,(\tau_1; \ldots; \tau_n; \epsilon)} \qquad \textsc{Proj}\ \frac{\Gamma \vdash M : \Pi\,(\tau_1; \ldots; \tau_i; R)}{\Gamma \vdash \mathit{proj}_i\ M : \tau_i}$$

These rules make sense with or without row variables

Projection does not care about the fields beyond i. Thanks to row variables, this can be expressed in terms of *parametric polymorphism:*

$$\mathit{proj}_i : \forall \alpha._1 \ldots \alpha_i \rho.\ \Pi\,(\alpha_1; \ldots; \alpha_i; \rho) \to \alpha_i$$

About Rows

Rows were invented by Wand and improved by RÃ©my in order to ascribe precise types to operations on *records.*

The case of tuples, presented here, is simpler.

Rows are used to describe *objects* in Objective Caml [Rémy and Vouillon, 1998].

Rows are explained in depth by Pottier and RÃ©my [Pottier and Rémy, 2005].

Closure-passing closure conversion

Rows and recursive types allow to define the translation of types in the closure-passing variant:

$$
\begin{array}{rll}
& [\![\tau_1 \to \tau_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
& \quad \mu\alpha. & \alpha \text{ is the concrete type of the closure} \\
& \quad\quad \Pi\,(& \text{a tuple...} \\
& \quad\quad\quad (\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; & \text{...that begins with a code pointer...} \\
& \quad\quad\quad \rho & \text{...and continues with the environment} \\
& \quad\quad) &
\end{array}
$$

See Morrisett and Harper's "fix-type" encoding [1998].

Question: Why is it $\exists\rho.\ \mu\alpha.\ \tau$ and not $\mu\alpha.\ \exists\rho.\ \tau$

The type of the environment is fixed once for all and does not change at each recursive call.

Question: Notice that ρ appears only once. Any comments?

Closure-passing closure conversion

Let $Clo(R)$ abbreviate $\mu\alpha.\Pi\ ((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R)$.

Let $UClo(R)$ abbreviate its unfolded version,
$\Pi\ ((Clo(R) \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R)$.

We have $[\![\tau_1 \to \tau_2]\!] = \exists\rho.Clo(\rho)$.

$$
\begin{array}{rcl}
[\![\lambda x{:}[\![\tau_1]\!].\, M]\!] & = & \textit{let } code : (Clo([\![\Gamma]\!]) \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
& & \quad \lambda(clo : Clo([\![\Gamma]\!], x : [\![\tau_1]\!]). \\
& & \quad\quad \textit{let } (\_, x_1, \ldots, x_n) : UClo[\![\Gamma]\!] = \textit{unfold } clo \textit{ in} \\
& & \quad\quad [\![M]\!] \textit{ in} \\
& & \textit{pack } [\![\Gamma]\!], (\textit{fold } (code, x_1, \ldots, x_n)) \\
& & \textit{as } \exists\rho.\, Clo(\rho) \\
\\
[\![M_1\ M_2]\!] & = & \textit{let } \rho, clo = \textit{unpack } [\![M_1]\!] \textit{ in} \\
& & \textit{let } code : (Clo(\rho) \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
& & \quad \textit{proj}_0\ (\textit{unfold } clo) \textit{ in} \\
& & code\ (clo, [\![M_2]\!])
\end{array}
$$

Closure-passing closure conversion recursive functions

In the closure-passing variant, recursive functions can be translated as:

$$
\begin{array}{rcl}
[\![\mu f.\lambda x.M]\!] & = & \textit{let } code = \lambda(clo, x). \\
& & \quad \textit{let } f = clo \textit{ in} \\
& & \quad \textit{let } (\_, x_1, \ldots, x_n) = clo \textit{ in} \\
& & \quad [\![M]\!] \\
& & \textit{in } (code, x_1, \ldots, x_n)
\end{array}
$$

where $\{x_1, \ldots, x_n\} = \mathrm{fv}(\mu f.\lambda x.M)$.

No extra field or extra work is required to store or construct a representation of the free variable f: the closure itself plays this role.

However, this untyped code can only be typechecked when recursion is monomorphic.

Exercise:

Check well-typedness with monomorphic recursion.

Closure-passing closure conversion recursive functions

The problem to adapt this encoding to polymorphic recursion is that recursive occurrences of f are rebuilt from the current invocation of the closure, *i.e.* is monomorphic since the closure is invoked after type specialization.

By contrast, in the environment passing encoding, the environment contained a polymorphic binding for the recursive calls that was filled with the closure before its invokation, *i.e.* with a polymorphic type.

Fortunately, we may slightly change the encoding, using a recursive closure as in the type-passing version, to allow typechecking in System F.

Closure-passing closure conversion recursive functions

Let τ be $\forall \vec{\alpha}.\, \tau_1 \to \tau_2$ and Γ_f be $f : \tau, \Gamma$ where $\vec{\beta} \# \Gamma$

$$
\begin{aligned}
[\![\mu f : \tau.\, \lambda x. M]\!] = \textit{let code} = \\
&\Lambda \vec{\beta}.\, \lambda(clo : Clo[\![\Gamma_f]\!], x : [\![\tau_1]\!]). \\
&\quad \textit{let}\ (\_code, f, x_1, \ldots, x_n) : \forall \vec{\beta}.\, UClo([\![\Gamma_f]\!]) = \\
&\qquad \textit{unfold}\ clo\ \textit{in} \\
&\quad [\![M]\!]\ \textit{in} \\
\textit{let rec}\ clo : \forall \vec{\beta}.\, \exists \rho.\, Clo(\rho) = \Lambda \vec{\beta}. \\
&\textit{pack}\ [\![\Gamma]\!], (\textit{fold}\ (code\ \vec{\beta}, clo, x_1, \ldots, x_n))\ \textit{as}\ \exists \rho.\, Clo(\rho) \\
\textit{in}\ clo
\end{aligned}
$$

Remind that $Clo(R)$ abbreviates $\mu\alpha.\Pi\ ((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R)$. Hence, $\vec{\beta}$ are free variables of $Clo(R)$.

Here, a polymorphic recursive function is *directly* compiled into a polymorphic recursive closure. Notice that the type of closures is unchanged so the encoding of applications is also unchanged.

Mutually recursive functions — Environment passing

Can we compile mutually recursive functions?

$$M \triangleq \mu(f_1, f_2).(\lambda x_1.\, M_1, \lambda x_2.\, M_2)$$

Environment passing:

$$
\begin{array}{rcl}
[\![M]\!] & = & \textit{let } code_i = \lambda(env, x). \\
& & \quad \textit{let } (f_1, f_2, x_1, \ldots, x_n) = env \textit{ in} \\
& & \quad [\![M_i]\!] \\
& & \textit{in} \\
& & \textit{let rec } clo_1 = (code_1, (clo_1, clo_2, x_1, \ldots, x_n)) \\
& & \quad \textit{and } clo_2 = (code_2, (clo_1, clo_2, x_1, \ldots, x_n)) \textit{ in} \\
& & clo_1, clo_2
\end{array}
$$

Mutually recursive functions — Environment passing

Can we compile mutually recursive functions?

$$M \triangleq \mu(f_1, f_2).(\lambda x_1.\, M_1, \lambda x_2.\, M_2)$$

Environment passing:

$$
\begin{array}{rcl}
[\![M]\!] & = & \textit{let}\ code_i = \lambda(env, x). \\
& & \quad \textit{let}\ (f_1, f_2, x_1, \ldots, x_n) = env\ \textit{in} \\
& & \quad [\![M_i]\!] \\
& & \textit{in} \\
& & \textit{let rec}\ env = (clo_1, clo_2, x_1, \ldots, x_n) \\
& & \quad \textit{and}\ clo_1 = (code_1, env) \\
& & \quad \textit{and}\ clo_2 = (code_2, env)\ \textit{in} \\
& & clo_1, clo_2
\end{array}
$$

Mutually recursive functions — Environment passing

Can we compile mutually recursive functions?

$$M \triangleq \mu(f_1, f_2).(\lambda x_1.\, M_1, \lambda x_2.\, M_2)$$

Environment passing:

$$\begin{array}{l}
\textit{let } code_i = \lambda(clo, x). \\
\quad \textit{let } (_\,, f_1, f_2, x_1, \ldots, x_n) = clo \textit{ in } [\![M_i]\!] \\
\textit{in} \\
\textit{let rec } clo_1 = (code_1, clo_1, clo_2, x_1, \ldots, x_n) \\
\quad \textit{and } clo_2 = (code_2, clo_1, clo_2, x_1, \ldots, x_n) \\
\textit{in } clo_1, clo_2
\end{array}$$

Question: Can we share the closures c_1 and c_2 in case n is large?

Mutually recursive functions

Environment passing

Can we compile mutually recursive functions?

$$M \triangleq \mu(f_1, f_2).(\lambda x_1.\, M_1, \lambda x_2.\, M_2)$$

Environment passing:

$$\begin{array}{l}
\textit{let } code_1 = \lambda(clo, x). \\
\quad \textit{let } (_code_1, _code_2, f_1, f_2, x_1, \ldots, x_n) = clo \textit{ in } [\![M_1]\!] \textit{ in} \\
\textit{let } code_2 = \lambda(clo, x). \\
\quad \textit{let } (_code_2, f_1, f_2, x_1, \ldots, x_n) = clo \textit{ in } [\![M_2]\!] \textit{ in} \\
\textit{let rec } clo_1 = (code_1, code_2, clo_1, clo_2, x_1, \ldots, x_n) \textit{ and } clo_2 = clo_1.tail \\
\textit{ in } clo_1, clo_2
\end{array}$$

- $clo_1.tail$ returns a pointer to the tail $(code_2, clo_1, clo_2, x_1, \ldots, x_n)$ of clo_1 without allocating a new tuple.
- This is only possible with some support from the GC (and extra-complexity and runtime cost for GC)

Optimizing representations

Can closure passing and environment passing be mixed?

No because the calling-convention (*i.e.*, the encoding of application) must be uniform.

However, their is some flexibility in the representation of the closure. For instance, the following change is completely local:

$$\begin{array}{rcl}
[\![\lambda x.\, M]\!] & = & \textit{let } code = \lambda(clo, x). \\
& & \quad \textit{let } (_, \boxed{(}\, x_1, \ldots, x_n \boxed{)}) = clo \textit{ in } [\![M]\!] \textit{ in} \\
& & (code, \boxed{(}\, x_1, \ldots, x_n \boxed{)}) \\
[\![M_1\ M_2]\!] & = & \textit{let } clo = [\![M_1]\!] \textit{ in} \\
& & \textit{let } code = \textit{proj}_0\ clo \textit{ in} \\
& & code\ (clo, [\![M_2]\!])
\end{array}$$

Applications? When many definitions share the same closure, the closure (or part of it) may be shared.

Encoding of objects

The closure-passing representation of mutually recursive functions is similar to the representations of objects in the object-as-record-of-functions paradigm:

A class definition is an object generator:

$$
\begin{array}{l}
\mathit{class}\ c\ (x_1, \ldots x_q) \{ \\
\qquad \textit{meth}\ \ m_1 = M_1 \\
\qquad \ldots \\
\qquad \textit{meth}\ \ m_p = M_p \\
\}
\end{array}
$$

Given arguments for parameter x_1, $\ldots x_1$, it will build recursive methods m_1, $\ldots m_n$.

Encoding of objects

A class can be compiled into an object closure:

$$
\begin{array}{l}
\textit{let } m = \\
\qquad \textit{let } m_1 = \lambda(m, x_1, \ldots, x_q).\, M_1 \textit{ in} \\
\qquad \ldots \\
\qquad \textit{let } m_p = \lambda(m, x_1, \ldots, x_q).\, M_p \textit{ in} \\
\qquad \{m_1, \ldots, m_p\} \;\; \textit{in} \\
\lambda x_1 \ldots x_q.\, (m, x_1, \ldots x_q)
\end{array}
$$

Each m_i is bound to the code for the corresponding method.
The code of all methods are combined into a record of methods, which is shared between all objects of the same class.

Calling method m_i of an object p is

$$(\textit{proj}_0\; p).m_i\; p$$

How can we type the encoding?

Typed encoding of objects

Let τ_i be the type of M_i, and row R describe the types of $(x_1, \dots x_q)$.

Let $Clo(R)$ be $\mu\alpha.\Pi(\{(m_i : \alpha \to \tau_i)^{i\in 1..n}\}; R)$ and $UClo(R)$ its unfolding.

Fields R are hidden in an existential type $\exists\rho.\,\mu\alpha.\Pi(\{(m_i : \alpha \to \tau_i)^{i\in I}\}; \rho)$:

$$
\begin{array}{l}
\textit{let}\; m = \{ \\
\quad m_1 = \lambda(m, x_1, \dots x_q : UClo(R)).\, [\![M_1]\!] \\
\quad \dots \\
\quad m_p = \lambda(m, x_1, \dots x_q : UClo(R)).\, [\![M_p]\!] \\
\}\;\; \textit{in} \\
\lambda x_1.\, \dots \lambda x_q.\, \textit{pack}\; R, \textit{fold}\; (m, x_1, \dots x_q)\; \textit{as}\; \exists\rho.\,(M, \rho)
\end{array}
$$

Calling a method of an object p of type M is

$$p\#m_i \triangleq \textit{let}\; \rho, z = \textit{unpack}\; p\; \textit{in}\; (\textit{proj}_0\; \textit{unfold}\; z).m_i\; z$$

An object has a recursive type but it is *not* a recursive value.

Typed encoding of objects

Typed encoding of objects were first studied in the 90's to understand what objects really are in a type setting.

These encodings are in fact type-preserving compilation of (primitive) objects.

There are several variations on these encodings. See [Bruce et al., 1999] for a comparison.

See [Rémy, 1994] for an encoding of objects in (a small extension of) ML with iso-existentials and universals.

See [Abadi and Cardelli, 1996, 1995] for more details on primitive objects.

Moral of the story

Type-preserving compilation is rather *fun.* (Yes, really!)

It forces compiler writers to make the structure of the compiled program *fully explicit,* in type-theoretic terms.

In practice, building explicit type derivations, ensuring that they remain small and can be efficiently typechecked, can be a lot of work.

Optimizations

Because we have focused on type preservation, we have studied only naÃ¯ve closure conversion algorithms.

More ambitious versions of closure conversion require program analysis: see, for instance, Steckler and Wand [1997]. These versions *can* be made type-preserving.

Other challenges

Defunctionalization, an alternative to closure conversion, offers an interesting challenge, with a simple solution [Pottier and Gauthier, 2006].

Designing an efficient, type-preserving compiler for an *object-oriented language* is quite challenging. See, for instance, Chen and Tarditi [2005].

Fomega: higher-kinds and higher-order types

Contents

Polymorphism in System F

Simply-typed λ-calculus

- no polymorphism
- many functions must be duplicated at different types

Via ML style (let-binding) polymorphism

- Considerable improvement by avoiding most of code duplication.
- ML has also local let-polymorphism (less critical).
- Still, ML is lacking existential types—compensated by modules and sometimes lacking higher-rank polymorphism

System F brings much more expressiveness

- Existential types—allows for type abstraction
- First-class universal types
- Allows for encoding of data structures and more programming patterns

Still, limited...

Limits of System F $\lambda f x y.\,(f\ x, f\ y)$

Map on pairs, say pair_map, has the following incompatible types:

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

The first one requires x and y to admit a common type, while the second one requires f to be polymorphic.

It is missing the ability to describe the types of functions

- that are polymorphic in one parameter
- but whose domain and codomain are otherwise arbitrary

i.e. of the form $\forall \alpha.\, \tau[\alpha] \to \sigma[\alpha]$ for arbitrary one-hole types τ and σ.

We just need to abstract over such contexts, *i.e.*, over *type functions*:

$$\forall \varphi\,.\,\forall \psi\,.\ \forall \alpha_1.\,\forall \alpha_2.\,(\forall \alpha.\ \varphi\ \alpha\ \to\ \psi\ \alpha\,) \to \varphi\ \alpha_1 \to \varphi\ \alpha_2 \to \psi\ \alpha_1 \times \psi\ \alpha_2$$

From System F to System F^ω — Kinds

We introduce kinds κ for types (with a single kind $*$ to stay in System F)

Well-formedness of types becomes $\Gamma \vdash \tau : *$:

$$\frac{\vdash \Gamma \quad \alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : * \quad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall \alpha :: \kappa . \tau : *}$$

$$\vdash \varnothing \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : * \quad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : \tau}$$

We add and check kinds on type abstractions and type applications:

$$\textsc{Tabs}\ \frac{\Gamma, \alpha : \kappa \vdash M : \tau}{\Gamma \vdash \Lambda \alpha :: \kappa . M : \forall \alpha :: \kappa . \tau} \qquad \textsc{Tapp}\ \frac{\Gamma \vdash M : \forall \alpha :: \kappa . \tau \quad \Gamma \vdash \tau' : \kappa}{\Gamma \vdash M\, \tau' : [\alpha \mapsto \tau'] \tau}$$

So far, this is an equivalent formalization of System F

From System F to System F^ω — Type functions

Redefine kinds as $\kappa ::= * \mid \kappa \Rightarrow \kappa$

New types $\tau ::= \dots \mid \lambda\alpha :: \kappa.\tau \mid \tau\ \tau$

$$\textsc{WfTypeApp}\quad \frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_1}$$

$$\textsc{WfTypeAbs}\quad \frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda\alpha :: \kappa_1.\tau : \kappa_1 \Rightarrow \kappa_2}$$

Typing of expressions is up to type equivalence:

$$\textsc{TConv}\quad \frac{\Gamma \vdash M : \tau \qquad \tau \equiv_\beta \tau'}{\Gamma \vdash M : \tau'}$$

Remark

$$\Gamma \vdash M : \tau \implies \Gamma \vdash \tau : *$$

F^ω, static semantics (altogether on one slide)

Syntax *With implicit kinds*

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

Kinding rules

$$\vdash \varnothing \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : \star \quad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : \tau} \qquad \frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : \star \quad \Gamma \vdash \tau_2 : \star}{\Gamma \vdash \tau_1 \to \tau_2 : \star}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \tau : \star}{\Gamma \vdash \forall \alpha.\,\tau : \star} \qquad \frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda\alpha.\,\tau : \kappa_1 \Rightarrow \kappa_2} \qquad \frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \quad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_1}$$

Typing rules

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\,M : \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\ M_2 : \tau_2}$$

$$\textsc{TAbs}\ \frac{\Gamma, \alpha : \kappa \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\,M : \forall \alpha.\tau} \qquad \textsc{TApp}\ \frac{\Gamma \vdash M : \forall \alpha.\tau \quad \Gamma \vdash \tau' : \kappa}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau} \qquad \textsc{TEquiv}\ \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash \tau \equiv_\beta \tau'}{\Gamma \vdash M : \tau'}$$

F^ω, dynamic semantics

The semantics is unchanged (modulo kind annotations in terms)

$$
\begin{array}{lcl}
V & ::= & \lambda x{:}\tau.\, M \mid \Lambda\alpha :: \kappa.\, V \\
E & ::= & [\,]\, M \mid V\, [\,] \mid [\,]\, \tau \mid \Lambda\alpha :: \kappa.\, [\,]
\end{array}
$$

$$
\begin{array}{c}
(\lambda x{:}\tau.\, M)\; V \longrightarrow [x \mapsto V]M \\
(\Lambda\alpha :: \kappa.\, V)\; \tau \longrightarrow [\alpha \mapsto \tau]V
\end{array}
$$

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

No type reduction

- We need not reduce types inside terms.
- β reduction on types is needed for type conversion (*i.e.* for typing) but such reduction need not be performed during term reduction.

Kinds are erasable

- Kinds are preserved by type and term reduction.
- Kinds may be ignored during reduction—or erased prior to reduction.

Properties

Main properties are preserved. Proofs are similar to those for System F.

Type soundness

- Subject reduction
- Progress

Termination of reduction

(In the absence of construct for recursion.)

Typechecking is decidable

- This requires reduction at the level of types to check type equality
- Can be done by putting types in normal forms using full reduction (on types only), or just head normal forms.

Type reduction

Used for typechecking to check type equivalence $\equiv$

Full reduction of the simply typed λ-calculus

$$(\lambda\alpha.\,\tau)\;\sigma \longrightarrow [\alpha \mapsto \tau]\sigma$$

applicable in *any type context*.

Type reduction preserve types: this is subject reduction for simply-typed λ-calculus (when terms are now used as types), but for *full reduction*

(we have only proved it for CBV).

It is a key that reduction terminates.

(which again, we have only proved for CBV.)

Contents

- Presentation
- Expressiveness
- Beyond F^ω

Expressiveness

More polymorphism

- pair_map

Abstraction over type operators

- monads
- encoding of existentials

Other encodings

- non regular datatypes
- equality
- *modules*

Pair map in F^ω (with implicit kinds) $\lambda f x y.\,(f\ x, f\ y)$

Abstract over (one parameter) type *functions* (*e.g.* of kind $\star \to \star$)

$$\Lambda\varphi.\,\Lambda\psi.\,\Lambda\alpha_1.\,\Lambda\alpha_2.$$
$$\lambda(f : \forall\alpha.\,\varphi\alpha \to \psi\alpha).\,\lambda x : \varphi\alpha_1.\,\lambda y : \varphi\alpha_2.\,(f\ \alpha_1\ x, f\ \alpha_2\ y)$$

call it pair_map of type:

$$\forall\varphi.\forall\psi.\forall\alpha_1.\forall\alpha_2.$$
$$(\forall\alpha.\,\varphi\alpha \to \psi\alpha) \to \varphi\alpha_1 \to \varphi\alpha_2 \to \psi\alpha_1 \times \psi\alpha_2$$

We may recover, in particular, the two types it has in System F:

$$\Lambda\alpha_1.\,\Lambda\alpha_2.\,\lambda f : \alpha_1 \to \alpha_2.\,\mathsf{pair\_map}\ (\lambda\alpha.\,\alpha_1)\ (\lambda\alpha.\,\alpha_2)\ \alpha_1\ \alpha_2\ (\Lambda\gamma.\,f)$$
$$: \forall\alpha_1.\,\forall\alpha_2.\,(\ \forall\gamma.\ \alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

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

Still, the type of pair_map is not principal: φ and ψ could depend on two variables, *i.e.* be of kind $* \Rightarrow * \Rightarrow *$, or many other kinds...

Pair map in F^ω (with implicit kinds) $\lambda fxy.\,(f\ x, f\ y)$

Abstract over (one parameter) type *functions* (*e.g.* of kind $\star \to \star$)

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

call it pair_map of type:

$$\forall\varphi.\forall\psi.\forall\alpha_1.\forall\alpha_2.\\ \quad (\forall\alpha.\,\varphi\alpha \to \psi\alpha) \to \varphi\alpha_1 \to \varphi\alpha_2 \to \psi\alpha_1 \times \psi\alpha_2$$

We may recover, in particular, the two types it has in System F:

$$\Lambda\alpha_1.\,\Lambda\alpha_2.\,\lambda f{:}\alpha_1 \to \alpha_2.\,\text{pair\_map}\ (\lambda\alpha.\,\alpha_1)\ (\lambda\alpha.\,\alpha_2)\ \alpha_1\ \alpha_2\ (\Lambda\gamma.\,f)\\ \quad : \forall\alpha_1.\,\forall\alpha_2.\,(\ \forall\gamma.\ \alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

$$\text{pair\_map}\ (\lambda\alpha.\alpha)\ (\lambda\alpha.\alpha)\\ \quad : \forall\alpha_1.\,\forall\alpha_2.\,(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

Still, the type of pair_map is not principal: φ and ψ could depend on two variables, *i.e.* be of kind $\ast \Rightarrow \ast \Rightarrow \ast$, or many other kinds...

Abstracting over type operators

Type of monads Given a type operator φ, a monad is given by a pair of two functions of the following type (satisfying certain laws).

$$
\begin{aligned}
\mathcal{M} \;&\triangleq\; \lambda\varphi.\\
&\qquad \{\; \mathit{ret} : \forall\alpha.\,\alpha \to \varphi\alpha;\\
&\qquad\;\; \mathit{bind} : \forall\alpha.\,\forall\beta.\,\varphi\alpha \to (\alpha \to \varphi\beta) \to \varphi\beta \;\}\\
&:\; (* \Rightarrow *) \Rightarrow *
\end{aligned}
$$

(Notice that $\mathcal{M}$ is itself of higher kind)

A generic map function: can then be defined:

$$
\begin{aligned}
&\mathit{fmap}\\
&\quad\triangleq\quad \lambda m.\\
&\qquad\qquad \lambda f.\,\lambda x.\\
&\qquad\qquad\qquad m.\mathit{bind}\, x\, (\lambda x.\, m.\mathit{ret}\, (f\; x))\\
&\quad:\quad \forall\varphi.\,\mathcal{M}\,\varphi \to \forall\alpha.\,\forall\beta.\,(\alpha \to \beta) \to \varphi\alpha \to \varphi\beta
\end{aligned}
$$

Abstracting over type operators

Type of monads Given a type operator φ, a monad is given by a pair of two functions of the following type (satisfying certain laws).

$$\begin{aligned}\mathcal{M} \;&\triangleq\; \lambda\varphi.\\ &\qquad \{\, \textit{ret} : \forall\alpha.\, \alpha \to \varphi\alpha;\\ &\qquad\;\; \textit{bind} : \forall\alpha.\, \forall\beta.\, \varphi\alpha \to (\alpha \to \varphi\beta) \to \varphi\beta \,\}\\ &: \; (\ast \Rightarrow \ast) \Rightarrow \ast\end{aligned}$$

(Notice that $\mathcal{M}$ is itself of higher kind)

A generic map function: can then be defined:

$$\begin{aligned}&\textit{fmap}\\ &\quad\triangleq\quad \lambda m.\\ &\qquad\qquad \lambda f.\, \lambda x.\\ &\qquad\qquad\qquad m.\textit{bind}\, x\, (\lambda x.\, m.\textit{ret}\, (f\ x))\\ &\quad:\quad \forall\varphi.\, \mathcal{M}\,\varphi \to \forall\alpha.\, \forall\beta.\, (\alpha \to \beta) \to \varphi\alpha \to \varphi\beta\end{aligned}$$

Abstracting over type operators

Available in Haskell —without β-reduction

- $\varphi\alpha$ is treated as a type $\mathsf{app}(\varphi, \alpha)$ where $\mathsf{app} : (\kappa_1 \Rightarrow \kappa_2) \Rightarrow \kappa_1 \Rightarrow \kappa_2$
- No β-reduction at the level of types: $\varphi\alpha = \psi\beta \iff \varphi = \psi \wedge \alpha = \beta$
- Compatible with type inference (first-order unification)
- Since there is no type β-reduction, this is not F^ω.

Encodable in OCaml with modules

- See [Yallop and White, 2014] (and also [Kiselyov])
- As in Haskell, the encoding does not handle type β-reduction
- As a counterpart, this allows for type inference at higher kinds (as in Haskell).

Encoding of existentials — Limits of System F

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;\triangleq\; \Lambda\alpha.\,\lambda x{:}[\![\tau]\!].\,\Lambda\beta.\,\lambda k{:}\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we just abstract over $\lambda\alpha.\,\tau$, *i.e.* φ:

In System F^ω, we may defined

$$[\![\mathit{pack}_{\kappa}]\!] \;=\; \Lambda\varphi.\,\Lambda\alpha.\,\lambda x:\varphi\;\alpha.\,\Lambda\beta.\,\lambda k:\forall\alpha.\,(\varphi\;\alpha \to \beta).\,k\;\alpha\;x$$

(omitting kinds)

Allows existentials at higher kinds!

Encoding of existentials — Limits of System F

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;\triangleq\; \Lambda\alpha.\,\lambda x{:}[\![\tau]\!].\,\Lambda\beta.\,\lambda k{:}\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we just abstract over $\lambda\alpha.\,\tau$, *i.e.* φ:

In System F^{ω}, we may defined

$$[\![\mathit{pack}_{\kappa}]\!] \;=\; \Lambda\varphi.\,\Lambda\alpha.\;\lambda x : \varphi\;\alpha.\,\Lambda\beta.\,\lambda k : \forall\alpha.\,(\varphi\;\alpha \to \beta).\,k\;\alpha\;x$$

(omitting kinds)

Allows existentials at higher kinds!

Exploiting kinds

Once we have type functions, the language of types could be reduced to λ-calculus with constants (plus arrow types kept as primitive):

$$\tau = \alpha \mid \lambda\alpha{:}\kappa.\,\tau \mid \tau\ \tau \mid \tau \to \tau \mid g$$

where type constants $g \in \mathcal{G}$ are given with their kind and syntactic sugar:

$$\begin{array}{lll}
\times & :: & * \Rightarrow * \Rightarrow * \\
+ & :: & * \Rightarrow * \Rightarrow \kappa \\
\forall_\kappa & :: & (\kappa \Rightarrow *) \Rightarrow * \\
\exists_\kappa & :: & (\kappa \Rightarrow *) \Rightarrow *
\end{array}
\qquad
\begin{array}{lll}
(\tau \times \tau) & \triangleq & (\times)\ \tau_1\ \tau_2 \\
(\tau + \tau) & \triangleq & (+)\ \tau_1\ \tau_2 \\
\forall\varphi : \kappa.\,\tau & \triangleq & \forall_\kappa(\lambda\varphi{:}\kappa \Rightarrow *.\,\tau) \\
\exists\varphi : \kappa.\,\tau & \triangleq & \exists_\kappa(\lambda\varphi{:}\kappa \Rightarrow *.\,\tau)
\end{array}$$

In fact F^ω could be extended with kind abdstraction:

$$\begin{array}{lll}
\hat\forall & :: & \forall\kappa.(\kappa \Rightarrow *) \Rightarrow * \\
\hat\exists & :: & \forall\kappa.(\kappa \Rightarrow *) \Rightarrow *
\end{array}
\qquad
\begin{array}{lll}
\forall\varphi : \kappa.\,\tau & \triangleq & \hat\forall\ \kappa\,(\lambda\varphi : \kappa \Rightarrow *.\,\tau) \\
\exists\varphi : \kappa.\,\tau & \triangleq & \hat\exists\ \kappa\,(\lambda\varphi : \kappa \Rightarrow *.\,\tau)
\end{array}$$

When kinds are inferred:

$$\begin{array}{lll}
\forall\varphi.\,\tau & \triangleq & \hat\forall\,(\lambda\varphi.\,\tau) \\
\exists\varphi.\,\tau & \triangleq & \hat\exists\,(\lambda\varphi.\,\tau)
\end{array}$$

Church encoding of regular ADT — List

$$
\begin{array}{ll}
\texttt{type} & \mathit{List}\ \alpha = \\
\quad \mid \mathit{Nil} & : \forall \alpha.\, \mathit{List}\ \alpha \\
\quad \mid \mathit{Cons} & : \forall \alpha.\, \alpha \to \mathit{List}\ \alpha \to \mathit{List}\ \alpha
\end{array}
$$

Church encoding (CPS style) in System F

$$
\begin{array}{lll}
\mathit{List} & \triangleq & \lambda \alpha.\, \forall \beta.\, \beta \to (\alpha \to \beta \to \beta) \to \beta \\
\mathit{Nil} & \triangleq & \lambda n.\, \lambda c.\, n \quad : \quad \forall \alpha.\, \mathit{List}\ \alpha \\
\mathit{Cons} & \triangleq & \lambda x.\, \lambda \ell.\, \lambda n.\, \lambda c.\, c\ x\ (\ell\ \beta\ n\ c) \quad : \quad \forall \alpha.\, \alpha \to \mathit{List}\ \alpha \to \mathit{List}\ \alpha
\end{array}
$$

$$
\mathit{fold} \triangleq \lambda n.\, \lambda c.\, \lambda \ell.\, \ell\ \beta\ n\ c
$$

Actually not enhanced !

Be aware of useless over-generalization!

For regular ADTs, all uses of φ are $\varphi\alpha$.

Hence, $\forall \alpha.\, \forall \varphi.\, \tau[\varphi\alpha]$ is not more general than $\forall \alpha.\, \forall \beta.\, \tau[\beta]$

Church encoding of regular ADT — List

$$
\begin{array}{lll}
\texttt{type} & \mathit{List}\ \alpha = & \\
\quad \mid \mathit{Nil} & : \forall\alpha.\,\mathit{List}\ \alpha & \\
\quad \mid \mathit{Cons} & : \forall\alpha.\,\alpha \to \mathit{List}\ \alpha \to \mathit{List}\ \alpha &
\end{array}
$$

Church encoding (CPS style) in System F

$$
\begin{array}{lll}
\mathit{List} & \triangleq & \boxed{\lambda\alpha.\,\forall\beta.\,\beta \to (\alpha \to \beta \to \beta) \to \beta} \\
\mathit{Nil} & \triangleq & \lambda n.\,\lambda c.\,n \quad : \quad \forall\alpha.\,\mathit{List}\ \alpha \\
\mathit{Cons} & \triangleq & \lambda x.\,\lambda\ell.\,\lambda n.\,\lambda c.\,c\ x\ (\ell\ \beta\ n\ c) \quad : \quad \forall\alpha.\,\alpha \to \mathit{List}\ \alpha \to \mathit{List}\ \alpha
\end{array}
$$

$$
\mathit{fold} \triangleq \lambda n.\,\lambda c.\,\lambda\ell.\,\ell\ \beta\ n\ c
$$

Actually not enhanced !

Be aware of useless over-generalization!

For regular ADTs, all uses of φ are $\varphi\alpha$.

Hence, $\forall\alpha.\,\forall\varphi.\,\tau[\varphi\alpha]$ is not more general than $\forall\alpha.\,\forall\beta.\,\tau[\beta]$

Church encoding of *non*-regular ADTs — Okasaki's Seq

$$
\begin{array}{ll}
\texttt{type} & \mathit{Seq}\,\alpha = \\
& \mid \mathit{Nil} : \forall\alpha.\,\mathit{Seq}\,\alpha \\
& \mid \mathit{Zero} : \forall\alpha.\,\mathit{Seq}\,(\alpha\times\alpha) \to \mathit{Seq}\,\alpha \\
& \mid \mathit{One} : \forall\alpha.\,\alpha \to \mathit{Seq}\,(\alpha\times\alpha) \to \mathit{Seq}\,\alpha
\end{array}
$$

Encoded as:

$$\mathit{Seq} \triangleq \lambda\alpha.\,\forall\varphi.\,(\forall\alpha.\,\varphi\,\alpha) \to (\forall\alpha.\,\varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to (\forall\alpha.\,\alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to \varphi\alpha$$

$$
\begin{array}{rcll}
\mathit{Nil} & \triangleq & \lambda n.\,\lambda z.\,\lambda s.\,n & :\ \forall\alpha.\ \mathit{Seq}\,\alpha \\
\mathit{Zero} & \triangleq & \lambda \ell.\,\lambda n.\,\lambda z.\,\lambda s.\,z\,(\ell\,n\,z\,s) & :\ \forall\alpha.\ \mathit{Seq}\,(\alpha\times\alpha) \to \mathit{Seq}\,\alpha \\
\mathit{One} & \triangleq & \lambda x.\,\lambda \ell.\,\lambda n.\,\lambda z.\,\lambda s.\,s\,x\,(\ell\,n\,z\,s) & :\ \forall\alpha.\ \alpha \to \mathit{Seq}\,(\alpha\times\alpha) \to \mathit{Seq}\,\alpha
\end{array}
$$

$$\mathit{fold} \triangleq \lambda n.\,\lambda z.\,\lambda s.\,\lambda \ell.\,\ell\,n\,z\,s$$

Cannot be simplified! Indeed φ is applied to both α and $\alpha \times \alpha$.

Non regular ADTs cannot be encoded in System F.

Church encoding of *non*-regular ADTs — Okasaki's Seq

$$
\begin{array}{l}
\texttt{type}\quad \mathit{Seq}\ \alpha = \\
\quad \mid \mathit{Nil} : \forall\alpha.\ \mathit{Seq}\ \alpha \\
\quad \mid \mathit{Zero} : \forall\alpha.\ \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha \\
\quad \mid \mathit{One} : \forall\alpha.\ \alpha \to \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha
\end{array}
$$

Encoded as:

$$\mathit{Seq} \triangleq \lambda\alpha.\ \forall\varphi.\ (\forall\alpha.\ \varphi\,\alpha) \to (\forall\alpha.\ \varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to (\forall\alpha.\ \alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to \varphi\alpha$$

$$
\begin{array}{rcll}
\mathit{Nil} & \triangleq & \lambda n.\ \lambda z.\ \lambda s.\ n & :\ \forall\alpha.\ \ \mathit{Seq}\ \alpha \\
\mathit{Zero} & \triangleq & \lambda \ell.\ \lambda n.\ \lambda z.\ \lambda s.\ z\ (\ell\ n\ z\ s) & :\ \forall\alpha.\ \ \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha \\
\mathit{One} & \triangleq & \lambda x.\ \lambda \ell.\ \lambda n.\ \lambda z.\ \lambda s.\ s\ x\ (\ell\ n\ z\ s) & :\ \forall\alpha.\ \ \alpha \to \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha
\end{array}
$$

$$\mathit{fold} \triangleq \lambda n.\ \lambda z.\ \lambda s.\ \lambda \ell.\ \ell\ n\ z\ s$$

Cannot be simplified! Indeed φ is applied to both α and $\alpha \times \alpha$.

Non regular ADTs cannot be encoded in System F.

Equality — Encoded with GADT

```
module Eq : EQ = struct
  type (α, β) eq = Eq : (α, α) eq

  let coerce (type a) (type b) (ab : (a,b) eq) (x : a) : b = let Eq = ab in x

  let refl : (α, α) eq = Eq

  (* all these are propagation and automatic with GADTs *)
  let symm (type a) (type b) (ab : (a,b) eq) : (b,a) eq = let Eq = ab in ab
  let trans (type a) (type b) (type c)
      (ab : (a,b) eq) (bc : (b,c) eq) : (a,c) eq = let Eq = ab in bc

  let lift (type a) (type b) (ab : (a,b) eq) : (a list, b list) eq =
    let Eq = ab in Eq

end
```

Equality

Leibnitz equality in F^ω

$\mathit{Eq}\ \alpha\ \beta \equiv \forall\varphi.\, \varphi\alpha \to \varphi\beta$

$$
\begin{array}{lcl}
\mathit{Eq} & \triangleq & \lambda\alpha.\,\lambda\beta.\,\forall\varphi.\,\varphi\alpha \to \varphi\beta \\
\mathit{coerce} & \triangleq & \lambda p.\,\lambda x.\ p\ x \\
 & : & \forall\alpha.\,\forall\beta.\,\mathit{Eq}\ \alpha\ \beta \to \alpha \to \beta \\
\mathit{refl} & \triangleq & \lambda x.\ x \\
 & : & \forall\alpha.\,\forall\varphi.\,\varphi\alpha \to \varphi\alpha \ \equiv\ \forall\alpha.\,\mathit{Eq}\ \alpha\ \alpha \\
\mathit{symm} & \triangleq & \lambda p.\ p\,(\mathit{refl}) \\
 & : & \forall\alpha.\,\forall\beta.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \alpha \qquad : \mathit{Eq}\ \alpha\ \alpha \to \mathit{Eq}\ \beta\ \alpha \\
\mathit{trans} & \triangleq & \lambda p.\,\lambda q.\ q\ p \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\gamma.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \gamma \to \mathit{Eq}\ \alpha\ \gamma \qquad : \mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \alpha\ \gamma \\
\mathit{lift} & \triangleq & \lambda p.\ p\,(\mathit{refl}) \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\varphi.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ (\varphi\alpha)\ (\varphi\beta) \qquad : \mathit{Eq}\ (\varphi\alpha)\ (\varphi\alpha) \to \mathit{Eq}\ (\varphi\alpha)\ (\varphi\beta)
\end{array}
$$

Equality

Leibniz equality in F^ω

$$\mathit{Eq}\ \alpha\ \beta \equiv \forall\varphi.\,\varphi\alpha \to \varphi\beta$$

$$\begin{array}{lcl}
\mathit{Eq} & \triangleq & \lambda\alpha.\,\lambda\beta.\,\forall\varphi.\,\varphi\alpha \to \varphi\beta \\
\mathit{coerce} & \triangleq & \lambda p.\,\lambda x.\ p\ x \\
 & : & \forall\alpha.\,\forall\beta.\,\mathit{Eq}\ \alpha\ \beta \to \alpha \to \beta \\
\mathit{refl} & \triangleq & \lambda x.\ x \\
 & : & \forall\alpha.\,\forall\varphi.\,\varphi\alpha \to \varphi\alpha \quad \equiv \quad \forall\alpha.\,\mathit{Eq}\ \alpha\ \alpha \\
\mathit{symm} & \triangleq & \lambda p.\ p\,(\mathit{refl}) \\
 & : & \forall\alpha.\,\forall\beta.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \alpha \qquad : \mathit{Eq}\ \alpha\ \alpha \to \mathit{Eq}\ \beta\ \alpha \\
\mathit{trans} & \triangleq & \lambda p.\,\lambda q.\ q\ p \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\gamma.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \gamma \to \mathit{Eq}\ \alpha\ \gamma \qquad : \mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \alpha\ \gamma \\
\mathit{lift} & \triangleq & \lambda p.\ p\,(\mathit{refl}) \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\varphi.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ (\varphi\alpha)\ (\varphi\beta) \qquad : \mathit{Eq}\ (\varphi\alpha)\ (\varphi\alpha) \to \mathit{Eq}\ (\varphi\alpha)\ (\varphi\beta)
\end{array}$$

Equality — Leibnitz equality in F^ω

We implemented parts of the coercions of System Fc.

- We do not have decomposition of equalities (the inverse of *Lift*).
- This requires injectivity of type operators, which is not given.
- Equivalences and liftings must be written explicitly, while they are implicit with GADTs.

Some GATDs can be encoded, using equality plus existential types.

Contents

A hierarchy of type systems

Kinds have a rank:

- the base kind $*$ is of rank 1
- kinds $* \Rightarrow *$ and $* \Rightarrow * \Rightarrow *$ have rank 2. They are the kinds of type functions taking type parameters of base kind.
- kind $(* \Rightarrow *) \Rightarrow *$ has rank 3—it is a type function whose parameter is itself a simple type function (of rank 1).
- more generally, *rank* $(\kappa_1 \Rightarrow \kappa_2) = \max(1 + \mathit{rank}\, \kappa_1, \mathit{rank}\, \kappa_2)$

This defines a sequence $F^1 \subseteq F^2 \subseteq F^3 \ldots \subseteq F^\omega$ of type systems of increasing expressiveness, where F^n only uses kinds of rank n, whose limit is F^ω and where System F is F^1.

(*Ranks are sometimes shifted by one, starting with* $F = F^2$.)

Most examples in practice (and those we wrote) are in F^2, just above F.

Extensions

Abstraction over kinds?

$$\forall(\varphi :: * \Rightarrow *).\forall(\psi :: * \Rightarrow *).\forall(\alpha_1 :: *).\forall(\alpha_2 :: *).$$
$$(\forall\,(\alpha :: *)\,.\,\varphi\alpha \to \psi\alpha) \to \varphi\alpha_1 \to \varphi\alpha_2 \to \psi\alpha_1 \times \psi\alpha_2$$

Motivation: pair_map does not have a principal type.

F^ω with several base kinds

We could have several base kinds, *e.g.* $*$ and *field* with type constructors:

$$\begin{aligned} \mathit{filled} &: * \Rightarrow \mathit{field} \\ \mathit{empty} &: \mathit{field} \end{aligned} \qquad \qquad \mathit{box} : \mathit{field} \Rightarrow *$$

Prevents ill-formed types such as $\mathit{box}\,(\alpha \to \mathit{filled}\,\alpha)$.

This allows to build values v of type $\mathit{box}\,\theta$ where θ of kind *field* statically tells whether v is *filled* with a value of type τ or *empty*.

Application:

This is used in OCaml for rows of object types, but kinds are hidden to the user:

let get (x : $\langle$ get : α; .. $\rangle$) : α = x#get

The dots ".." here stand for a variable of another base kind (representing a *row* of types).

System F^ω with equirecursive types

Checking equality of equirecursive types in System F is already non obvious, since unfolding may require α-conversion to avoid variable capture. (See also [Gauthier and Pottier, 2004].)

With higher-order types, it is even trickier, since unfolding at functional kinds could expose new type redexes.

Besides, the language of types would be the simply type λ-calculus with a fix-point operator: type reduction would not terminate.

Therefore type equality would be undecidable, as well as type checking.

A solution is to restrict to recursion at the base kind $\star$. This allows to define recursive types but not recursive type functions.

Such an extension has been proven sound and and decidable, but only for the weak form or equirecursive types (with the unfolding but not the uniqueness rule)—see [Cai et al., 2016].

System F^ω with equirecursive kinds

Instead, recursion could also occur just at the level of kinds, allowing kinds to be themselves recursive.

Then, the language of types is the simply type λ-calculus with recursive types, equivalent to the untyped λ-calculus—every term is typable. Reduction of types does not terminate and type equality is ill-defined.

A solution proposed by Pottier [2011] is to force recursive kinds to be productive, reusing an idea from an [Nakano, 2000, 2001] for controlling recursion on terms, but pushing it one level up. Type equality becomes well-defined and semi-decidable.

The extension has been used to show that references in System F can be translated away in F^ω with guarded recursive kinds.

Encoding ML modules with *generative* functors

Generative functors can be encoded with existential types.

A functor F has a type of the form:

$$\forall \bar\alpha.\, \tau[\bar\alpha] \rightarrow \exists \bar\beta.\, \sigma[\bar\alpha, \bar\beta]$$

Where:

- $\tau[\bar\alpha]$ represents the signature of the argument with some abstract types $\bar\alpha$.
- $\exists \bar\beta.\, \sigma[\bar\alpha, \bar\beta]$ represents the signature of the result of the functor application.
- That is, the abstract types $\bar\alpha$ are those taken from and shared with the argument.
- Conversely $\bar\beta$ are the abstract types created by the application, and have fresh identities independent of the argument.
- Two successive applications with the *same* argument (hence the same α) will create two signatures with incompatible abstract types $\bar\beta_1$ and $\bar\beta_2$, once the existential is open.

Two applications of F with the same argument: must be understood as:

Encoding ML modules with *applicative* functors

Applicative functors can be encoded with *higher-order* existential types.

A functor F has a type of the form:

$$\exists \bar{\varphi}. \forall \bar{\alpha}. \tau[\bar{\alpha}] \to \exists \bar{\beta}. \sigma[\bar{\alpha}, \ \bar{\beta}]$$

Compared with:

$$\exists \varphi. \forall \bar{\alpha}. \tau[\bar{\alpha}] \to \exists \bar{\beta}. \sigma[\bar{\alpha}, \ \bar{\beta}]$$

That is:

- $\sigma[\bar{\alpha}, \bar{\varphi}\bar{\alpha}]$ represents the signature of the result of the functor application.
- $\bar{\varphi}\bar{\alpha}$ are the abstract types created by the application. Each $\varphi\bar{\alpha}$ is a new abstract type—one we know nothing about, as it is the application of an abstract type to $\bar{\alpha}$.
- However, two successive applications with the *same* argument (hence the same $\bar{\alpha}$) will create two *compatible* structures whose signatures have the same *shared* abstract types $\bar{\varphi}\bar{\alpha}$.

The two applications of F:

becomes:

let $\bar{\varphi}$, F = unpack F **in**

System F^ω in OCaml

Second-order polymorphism in OCaml

- Via polymorphic methods

  let id = **object method** f : α. $\alpha \to \alpha$ = **fun** x → x end
  let y (x : ⟨f : α. $\alpha \to \alpha$⟩) = x#f x **in** y id

- Via first-class modules

  module type S = sig **val** f : $\alpha \to \alpha$ end
  let id = (**module** struct **let** f x = x end : S)
  let y (x : (**module** S)) = **let module** X = (**val** x) **in** X.f x **in** y id

Higher-order types in OCaml

- In principle, they could be encoded with first-class modules.
- Not currently possible, due to (unnecessary) restrictions.
- Modular explicits, an extension that allows a better integration of abstraction over first-class modules will remove these limitations and allow a light-weight encoding of F^ω—with boiler-plate glue code.

System F^ω in OCaml

Second-order polymorphism in OCaml

- Via polymorphic methods

  let id = **object method** f : α. $\alpha \to \alpha$ = **fun** x → x end
  let y (x : ⟨f : α. $\alpha \to \alpha$⟩) = x#f x **in** y id

- Via first-class modules

  module type S = sig **val** f : $\alpha \to \alpha$ end
  let id = (**module** struct **let** f x = x end : S)
  let y (x : (**module** S)) = **let module** X = (**val** x) **in** X.f x **in** y id

Higher-order types in OCaml

- In principle, they could be encoded with first-class modules.
- Not currently possible, due to (unnecessary) restrictions.
- Modular explicits, an extension that allows a better integration of abstraction over first-class modules will remove these limitations and allow a light-weight encoding of F^ω—with boiler-plate glue code.

System F^ω in OCaml . . . with modular explicits

Available at git@github.com:mrmr1993/ocaml.git

```
module type s = sig type t end
module type op = functor (A:s) → s

let dp {F:op} {G:op} {A:s} {B:s} (f:{C:s} → F(C).t → G(C).t)
    (x : F(A).t) (y : F(B).t) : G(A).t * G(B).t = f {A} x, f {B} y
```

And its two specialized versions:

```
let dp1 (type a) (type b) (f : {C:s} → C.t → C.t) : a → b → a * b =
  let module F(C:s) = C in let module G = F in
  let module A = struct type t = a end in
  let module B = struct type t = b end in
  dp {F} {G} {A} {B} f

let dp2 (type a) (type b) (f : a → b) : a → a → b * b =
  let module A = struct type t = a end in
  let module B = struct type t = b end in
  let module F(C:s) = A in let module G(C:s) = B in
  dp {F} {G} {A} {B} (fun {C:s} → f)
```

System F^ω in Scala-3

Higher-order polymorphism a la System F^ω is available in Scala-3.

The monad example (with some variation on the signature) is:

```
trait Monad [F[_]] {
  def pure [A] (x: A) : F[A]
  def flatMap [A, B] (fa: F[A]) (f: A ⇒ F[B]) : F[B]
}
```

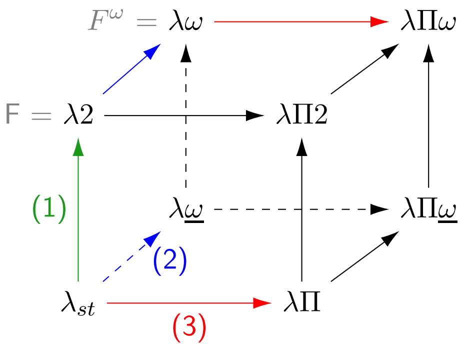
See https://www.baeldung.com/scala/dotty-scala-3

Still, this feature of Scala-3 is not emphasized

- It was not directly available in previous versions of Scala.
- Scala's syntax and other complex features of Scala are obfuscating.

What's next? Dependent types!

Barendregt's λ-cube



(1) Term abstraction on Types (example: System F)

(2) Type abstraction on Types (example: F^ω)

(3) *Type abstraction on Terms (dependent types)*

Logical relations and parametricity

Contents

What are logical relations?

So far, most proofs involving terms have proceeded by induction on the structure of *terms* (or, equivalently, on *typing derivations*).

Logical relations are relations between well-typed terms defined inductively on the structure of *types*. They allow proofs between terms by induction on the structure of *types*.

Unary relations

- Unary relations are predicates on expressions (or sets of expressions)
- They can be used to prove type safety and strong normalization

Binary relations

- Binary relations relate pairs of expressions of related types
- They can be used to prove equivalence of programs and non-interference properties.

Logical relations are a common proof method for programming languages.

Parametricity? Inhabitants of polymorphic types

In the presence of polymorphism (and in the absence of effects), a type can reveal a lot of information about the terms that inhabit it.

What can do a term of type $\forall\alpha.\,\alpha \to \mathit{int}$?

- the function cannot examine its argument
- it always returns the same integer
- $\lambda x.\,n$,
  $\lambda x.\,(\lambda y.\,y)\;n$,
  $\lambda x.\,(\lambda y.\,n)\;x$.
  etc.
- they are all $\beta\eta$-equivalent to the term $\lambda x.\,n$

Parametricity? Inhabitants of polymorphic types

In the presence of polymorphism (and in the absence of effects), a type can reveal a lot of information about the terms that inhabit it.

A term of type $\forall\alpha.\,\alpha \to \mathit{int}$?

- behaves as $\lambda x.\,n$

A term a of type $\forall\alpha.\,\alpha \to \alpha$?

- behaves as $\lambda x.\,x$

A term type $\forall\alpha\beta.\,\alpha \to \beta \to \alpha$?

- behaves as $\lambda x.\,\lambda y.\,x$

A term type $\forall\alpha.\,\alpha \to \alpha \to \alpha$?

- behaves either as $\lambda x.\,\lambda y.\,x$ or $\lambda x.\,\lambda y.\,y$

Pametricity

Theorems for free

Similarly, the type of a polymorphic function may also reveal a *"free theorem"* about its behavior!

What properties may we learn from a function

$$\texttt{whoami} \ : \ \forall\alpha.\,\mathit{list}\,\alpha \to \mathit{list}\,\alpha$$

- The length of the result depends only on the length of the argument
- All elements of the results are elements of the argument
- The choice (i, j) of pairs such that i-th element of the result is the j-th element of the argument does not depend on the element itself.
- the function is preserved by a transformation of its argument that preserves the shape of the argument

$$\forall f, x, \quad \texttt{whoami}\ (map\ f\ x) = map\ f\ (\texttt{whoami}\ x)$$

Pametricity — Theorems for free

Similarly, the type of a polymorphic function may also reveal a *"free theorem"* about its behavior!

What properties may we learn from a function

$$\texttt{whoami} \;:\; \forall\alpha.\, \mathit{list}\,\alpha \to \mathit{list}\,\alpha$$

What property may we learn for the list sorting function?

$$\mathit{sort} \;:\; \forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha \to \mathit{list}\,\alpha$$

If f is order-preserving, then sorting commutes with $map\ f$

$$(\forall x, y,\;\; \mathit{cmp}\,(f\ x)\,(f\ y) = \mathit{cmp}\ x\ y) \implies \forall \ell,\;\; \mathit{sort}\;\; \mathit{cmp}\,(\mathit{map}\ f\ \ell) = \mathit{map}\ f\ (\mathit{sort}\ \mathit{cmp}\ \ell)$$

Pametricity

Theorems for free

Similarly, the type of a polymorphic function may also reveal a *"free theorem"* about its behavior!

What properties may we learn from a function

$$\texttt{whoami} \;:\; \forall \alpha.\, \mathit{list}\, \alpha \to \mathit{list}\, \alpha$$

What property may we learn for the list sorting function?

$$\mathit{sort} \;:\; \forall \alpha.\, (\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\, \alpha \to \mathit{list}\, \alpha$$

If f is order-preserving, then sorting commutes with $map\ f$

$$(\forall x, y,\ \ \mathit{cmp}_2\ (f\ x)\ (f\ y) = \mathit{cmp}_1\ \ x\ y) \implies \\ \forall \ell,\ \ \mathit{sort}\ \ \mathit{cmp}_2\ (\mathit{map}\ f\ \ell) = \mathit{map}\ f\ (\mathit{sort}\ \mathit{cmp}_1\ \ell)$$

Application:

- If *sort* is correct on lists of integers, then it is correct on any list
- May be useful to reduce testing.

Pametricity

Theorems for free

Similarly, the type of a polymorphic function may also reveal a *"free theorem"* about its behavior!

What properties may we learn from a function

$$\texttt{whoami} \;:\; \forall\alpha.\, \mathit{list}\,\alpha \to \mathit{list}\,\alpha$$

What property may we learn for the list sorting function?

$$\mathit{sort} \;:\; \forall\alpha.\, (\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha \to \mathit{list}\,\alpha$$

If f is order-preserving, then sorting commutes with $map\ f$

$$(\forall x, y,\;\; \mathit{cmp}_2\ (f\ x)\ (f\ y) = \mathit{cmp}_1\ \ x\ y) \implies$$
$$\forall \ell,\;\; \mathit{sort}\ \ \mathit{cmp}_2\ (\mathit{map}\ f\ \ell) = \mathit{map}\ f\ (\mathit{sort}\ \mathit{cmp}_1\ \ell)$$

Note that there are many other inhabitants of this type, but they all satisfy this free theorem. (e.g., a function that sorts in reverse order, or a function that removes (or adds) duplicates).

Parametricity

This phenomenon was studied by Reynolds [1983] and by Wadler [1989; 2007], among others. Wadler's paper contains the 'free theorem' about the list sorting function.

An account based on an operational semantics is offered by Pitts [2000].

Bernardy et al. [2010] generalize the idea of testing polymorphic functions to arbitrary polymorphic types and show how testing any function can be restricted to testing it on (possibly infinitely many) particular values at some particular types.

Contents

Normalization of simply-typed λ-calculus

Types usually ensure termination of programs—as long as neither types nor terms contain any form of recursion.

Even if one wishes to add recursion explicitly later on, it is an important property of the design that non-termination is originating from the constructions introduced especially for recursion and could not occur without them.

The simply-typed λ-calculus is also lifted at the level of types in richer type systems such as F^ω; then, the decidability of type-equality depends on the termination of the reduction at the type level.

The proof of termination for the simply-typed λ-calculus is a simple and illustrative use of logical relations.

Notice however, that our simply-typed λ-calculus is equipped with a call-by-value semantics. Proofs of termination are usually done with a strong evaluation strategy where reduction can occur in any context.

Normalization

Proving termination of reduction in fragments of the λ-calculus is often a difficult task because reduction may create new redexes or duplicate existing ones.

Hence the size of terms may grow (much) larger during reduction. The difficulty is to find some underlying structure that decreases.

We follow the proof schema of Pierce [2002], which is a modern presentation in a call-by-value setting of an older proof by Hindley and Seldin [1986]. The proof method is due to [Tait, 1967].

Tait's method

Idea

- build the set $\mathcal{T}_\tau$ of terminating terms of type τ;
- show that any term of type τ is in $\mathcal{T}_\tau$, by induction on terms.

This hypothesis is however too weak. The difficulty is as usual to find a strong enough induction hypothesis...

Terms of type $\tau_1 \to \tau_2$ should not only terminate but also terminate when applied to terms in $\mathcal{T}_{\tau_1}$.

The construction of $\mathcal{T}_\tau$ is thus by induction of τ.

Normalization

Definition

Let $\mathcal{T}_\tau$ be defined inductively on τ as follows:

- $\mathcal{T}_\alpha$ is the set of closed terms that terminates;
- $\mathcal{T}_{\tau_2 \to \tau_1}$ is the set of closed terms M_1 of type $\tau_2 \to \tau_1$ that terminates and such that $M_1\ M_2$ is in $\mathcal{T}_{\tau_1}$ for any term M_2 in $\mathcal{T}_{\tau_2}$.

The set $\mathcal{T}_\tau$ can be seen as a predicate, *i.e.* a unary relation. It is called a *logical relation* because it is defined *inductively on the structure of types*.

The following proofs is then schematic of the use of logical relations.

Normalization

Reduction of terms of type τ preserves membership in $\mathcal{T}_\tau$ (this is stronger that stability of $\mathcal{T}_\tau$ by reduction):

Lemma

If $\varnothing \vdash M : \tau$ and $M \longrightarrow M'$, then $M \in \mathcal{T}_\tau$ iff $M' \in \mathcal{T}_\tau$.

Proof.

The proof is by induction on τ. □

Lemma

For any type τ, the reduction of any term in $\mathcal{T}_\tau$ terminates.

Tautology, by definition of $\mathcal{T}_\tau$.

Normalization

Therefore, it just remains to show that any term of type τ is in $\mathcal{T}_\tau$, *i.e.*:

Lemma

If $\varnothing \vdash M : \tau$, then $M \in \mathcal{T}_\tau$.

The proof is by induction on (the typing derivation of) M.

However, the case for abstraction requires some similar statement, but for open terms. We need to strengthen the Lemma.

A trick to avoid considering open terms is to require the statement to hold for all closed instances of an open term:

Lemma (strenghened)

If $(x_i : \tau_i)^{i \in I} \vdash M : \tau$, then for any closed values $(V_i)^{i \in I}$ in $(\mathcal{T}_{\tau_i})^{i \in I}$, the term $[(x_i \mapsto V_i)^{i \in I}]M$ is in $\mathcal{T}_\tau$.

Normalization

Proof. By structural induction on M.
We write Γ for $(x_i : \tau_i)^{i \in I}$ and θ for $[(x_i \mapsto V_i)^{i \in I}]$. Assume $\Gamma \vdash M : \tau$.

The only interesting case is when$M$ is $\lambda x{:}\tau_1.\, M_2$:

By inversion of typing, we know that $\Gamma, x : \tau_1 \vdash M_2 : \tau_2$ and $\tau_1 \to \tau_2$ is τ.

To show $\theta M \in \mathcal{T}_\tau$, we must show that it is terminating, which holds as it is a value, and that its application to any M_1 in $\mathcal{T}_{\tau_1}$ is in $\mathcal{T}_{\tau_2}$ $(\mathbf{1})$.

Let $M_1 \in \mathcal{T}_{\tau_1}$. By definition $M_1 \longrightarrow^* V$ $(\mathbf{2})$. We then have:

$$
\begin{array}{rcll}
(\theta M)\, M_1 & \triangleq & (\theta(\lambda x{:}\tau_1.\, M_2))\, M_1 & \text{by definition of } M \\
& = & (\lambda x{:}\tau_1.\, \theta M_2)\, M_1 & \text{choose } x \# \vec{x} \\
& \longrightarrow^* & (\lambda x{:}\tau_1.\, \theta M_2)\, V & \text{by } (2) \\
& \longrightarrow & [x \mapsto V](\theta M_2) & \text{by } (\beta) \\
& = & ([x \mapsto V]\theta)(M_2) \;\in\; \mathcal{T}_{\tau_2} & \text{by induction hypothesis}
\end{array}
$$

This establishes (1) since membership in $\mathcal{T}_{\tau_2}$ is preserved by reduction.

Calculus

Take the call-by-value λ_{st} with primitive booleans and conditional.

Write B the type of booleans and tt and ff for *true* and *false*.

We define $\mathcal{V}[\![\tau]\!]$ and $\mathcal{E}[\![\tau]\!]$ the subsets of closed values and closed expressions of (ground) type τ by induction on types as follows:

$$\begin{aligned}
\mathcal{V}[\![\mathsf{B}]\!] &\triangleq \{\mathsf{tt}, \mathsf{ff}\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!] &\triangleq \{\lambda x\!:\!\tau_1.\, M \mid \forall V \in \mathcal{V}[\![\tau_1]\!],\ (\lambda x\!:\!\tau_1.\, M)\; V \in \mathcal{E}[\![\tau_2]\!]\} \\
\mathcal{E}[\![\tau]\!] &\triangleq \{M \mid \exists V \in \mathcal{V}[\![\tau]\!], M \Downarrow V\}
\end{aligned}$$

We write $M \Downarrow N$ for $M \longrightarrow^* N$.
The goal is to show that any closed expression of type τ is in $\mathcal{E}[\![\tau]\!]$.

Remarks

Although usual with logical relations, well-typedness is actually not required here and omitted: otherwise, we would have to carry unnecessary type-preservation proof obligations. $\mathcal{V}[\![\tau]\!] \subseteq \mathcal{E}[\![\tau]\!]$—by definition.
$\mathcal{E}[\![\tau]\!]$ is closed by inverse reduction—by definition, *i.e.*
If $M \Downarrow N$ and $N \in \mathcal{E}[\![\tau]\!]$ then $M \in \mathcal{E}[\![\tau]\!]$

Problem

We wish to show that every closed term of type τ is in $\mathcal{E}[\![\tau]\!]$

- Proof by induction on the typing derivation.
- Problem with abstraction: the premise is not closed.

We need to strengthen the hypothesis, *i.e.* also *give a semantics to open terms*.

- *The semantics of open terms* can be given by *abstracting over the semantics of their free variables.*

Generalize the definition to open terms

We define a *semantic judgment* for open terms $\Gamma \vDash M : \tau$ so that $\Gamma \vdash M : \tau$ implies $\Gamma \vDash M : \tau$ and $\varnothing \vDash M : \tau$ means $M \in \mathcal{E}[\![\tau]\!]$.

We interpret free term variables of type τ as *closed values* in $\mathcal{V}[\![\tau]\!]$.

We interpret environments Γ as *closing substitutions* γ, *i.e.* mappings from term variables to *closed values*:

We write $\gamma \in \mathcal{G}[\![\Gamma]\!]$ to mean $\mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma)$ and $\gamma(x) \in \mathcal{V}[\![\tau]\!]$ for all $x : \tau \in \Gamma$.

$$\Gamma \vDash M : \tau \overset{\text{def}}{\iff} \forall \gamma \in \mathcal{G}[\![\Gamma]\!],\ \ \gamma(M) \in \mathcal{E}[\![\tau]\!]$$

Fundamental Lemma

Theorem (fundamental lemma)
If $\Gamma \vdash M : \tau$ then $\Gamma \vDash M : \tau$.

Corollary (termination of well-typed terms):
If $\varnothing \vdash M : \tau$ then $M \in \mathcal{E}[\![\tau]\!]$.

That is, closed well-typed terms of type τ evaluate to values of type τ.

Proof by induction on the typing derivation

Routine cases

Case $\Gamma \vdash \mathsf{tt} : \mathsf{B}$ *or* $\Gamma \vdash \mathsf{ff} : \mathsf{B}$: by definition, $\mathsf{tt}, \mathsf{ff} \in \mathcal{V}[\![\mathsf{B}]\!]$ and $\mathcal{V}[\![\mathsf{B}]\!] \subseteq \mathcal{E}[\![\mathsf{B}]\!]$.

Case $\Gamma \vdash x : \tau$: $\gamma \in \mathcal{G}[\![\Gamma]\!]$, thus $\gamma(x) \in \mathcal{V}[\![\tau]\!] \subseteq \mathcal{E}[\![\tau]\!]$

Case $\Gamma \vdash M_1\ M_2 : \tau$:

By inversion, $\Gamma \vdash M_1 : \tau_2 \to \tau$ and $\Gamma \vdash M_2 : \tau_2$.

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. We have $\gamma(M_1\ M_2) = (\gamma M_1)\ (\gamma M_2)$.
By IH, we have $\Gamma \vDash M_1 : \tau_2 \to \tau$ and $\Gamma \vDash M_2 : \tau_2$.
Thus $\gamma M_1 \in \mathcal{E}[\![\tau_2 \to \tau]\!]$ $(\mathbf{1})$ and $\gamma M_2 \in \mathcal{E}[\![\tau_2]\!]$ $(\mathbf{2})$.

By (2), there exists $V \in \mathcal{V}[\![\tau_2]\!]$ such that $\gamma M_2 \Downarrow V$.
Thus $(\gamma M_1)\ (\gamma M_2) \Downarrow (\gamma M_1)\ V \in \mathcal{E}[\![\tau]\!]$ by (1).

Then, $(\gamma M_1)\ (\gamma M_2) \in \mathcal{E}[\![\tau]\!]$, by closure by inverse reduction.

Case $\Gamma \vdash$ *if* M *then* M_1 *else* $M_2 : \tau$: By cases on the evaluation of γM.

Proof by induction on the typing derivation (key case)

The interesting case

Case $\Gamma \vdash \lambda x{:}\tau_1.\, M : \tau_1 \to \tau$:

Assume $\gamma \in \mathcal{G}[\![\Gamma]\!]$.
We must show that $\gamma(\lambda x{:}\tau_1.\, M) \in \mathcal{E}[\![\tau_1 \to \tau]\!]$ (1)

That is, $\lambda x{:}\tau_1.\, \gamma M \in \mathcal{V}[\![\tau_1 \to \tau]\!]$ (we may assume $x \notin \mathrm{dom}(\gamma)$ *w.l.o.g.*)

Let $V \in \mathcal{V}[\![\tau_1]\!]$, it suffices to show $(\lambda x{:}\tau_1.\, \gamma M)\ V \in \mathcal{E}[\![\tau]\!]$ (2).

We have $(\lambda x{:}\tau_1.\, \gamma M)\ V \longrightarrow (\gamma M)[x \mapsto V] = \gamma' M$
where γ' is $\gamma[x \mapsto V] \in \mathcal{G}[\![\Gamma, x : \tau_1]\!]$ (3)

Since $\Gamma, x : \tau_1 \vdash M : \tau$, we have $\Gamma, x : \tau_1 \vDash M : \tau$ by IH on M. Therefore by (3), we have $\gamma' M \in \mathcal{E}[\![\tau]\!]$. Since $\mathcal{E}[\![\tau]\!]$ is closed by inverse reduction, this proves (2) which finishes the proof of (1).

Variations

We have shown both *termination* and *type soundness*, simultaneously.

Termination would not hold if we had a fix point.
But type soundness would still hold.

The proof may be modified by choosing:

$$\mathcal{E}[\![\tau]\!] = \left\{M : \tau \mid \forall N, M \Downarrow N \implies \left(N \in \mathcal{V}[\![\tau]\!] \lor \exists N', N \longrightarrow N'\right)\right\}$$

Compare with

$$\mathcal{E}[\![\tau]\!] = \{M : \tau \mid \exists V \in \mathcal{V}[\![\tau]\!], M \Downarrow V\}$$

Exercise

Show type soundness with this semantics.

Contents

(Bibliography)

Mostly following Bob Harper's course notes *Practical foundations for programming languages* [Harper, 2012].

See also

- *Types, Abstraction and Parametric Polymorphism* [Reynolds, 1983]
- *Parametric Polymorphism and Operational Equivalence* [Pitts, 2000].
- *Theorems for free!* [Wadler, 1989].
- Course notes taken by Lau Skorstengaard on Amal Ahmed's OPLSS lectures.

We assume a call-by-value operational semantics instead of call-by-name in [Harper, 2012].

When are two programs equivalent

$M \Downarrow N$?

$M \Downarrow V$ and $N \Downarrow V$?

But what if M and N are functions?

Aren't $\lambda x.\,(x + x)$ and $\lambda x.\,2 * x$ equivalent?

Idea two functions are observationally equivalent if when applied to *equivalent arguments*, they lead to observationally *equivalent results*.

Are we general enough?

Observational equivalence

We can only *observe* the behavior of full *programs*, *i.e.* closed terms of some computation type, such as B (the only one so far).

If $M : \mathsf{B}$ and $N : \mathsf{B}$, then $M \simeq N$ iff there exists V such that $M \Downarrow V$ and $N \Downarrow V$. (Call $M \simeq N$ *behavioral equivalence*.)

To compare programs at other types, we place them in arbitrary *closing contexts*.

Definition (observational equivalence)

$$\Gamma \vdash M \cong N : \tau \quad \triangleq \quad \forall \mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\varnothing \triangleright \mathsf{B}),\ \mathcal{C}[M] \simeq \mathcal{C}[N]$$

Typing of contexts

$$\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Delta \triangleright \sigma) \iff (\forall M,\ \Gamma \vdash M : \tau \implies \Delta \vdash \mathcal{C}[M] : \sigma)$$

There is an equivalent definition given by a set of typing rules. This is needed to prove some properties by induction on the typing derivations.

We write $M \cong_\tau N$ for $\varnothing \vdash M \cong N : \tau$

Observational equivalence

Observational equivalence is the coarsiest consistent congruence, where:

- $\equiv$ is consistent if $\varnothing \vdash M \equiv N : \mathsf{B}$ implies $M \simeq N$.
- $\equiv$ is a congruence if it is an equivalence and is closed by context, *i.e.*

$$\Gamma \vdash M \equiv N : \tau \ \wedge\ \mathcal{C} : (\Gamma \triangleright \tau) \leadsto (\Delta \triangleright \sigma) \quad \Longrightarrow \quad \Delta \vdash \mathcal{C}[M] \equiv \mathcal{C}[N] : \sigma$$

Consistent: by definition, using the empty context.

Congruence: by compositionality of contexts.

Coarsiest: Assume $\equiv$ is a consistent congruence.

We assume $\Gamma \vdash M \equiv N : \tau$ (1) and show $\Gamma \vdash M \cong N : \tau$.

Let $\mathcal{C} : (\Gamma \triangleright \tau) \leadsto (\varnothing \triangleright \mathsf{B})$ (2). We must show that $\mathcal{C}[M] \simeq \mathcal{C}[N]$.
This follows by consistency applied to $\Gamma \vdash \mathcal{C}[M] \equiv \mathcal{C}[N] : \mathsf{B}$
which itself follows by congruence from (1) and (2).

Problem with Observational Equivalence

Problems

- Observational equivalence is too difficult to test.
- Because of quantification over all contexts (too many for testing).
- But many contexts will do the same experiment.

Solution

We take advantage of types to reduce the number of experiments.

- Defining/testing the equivalence on base types.
- Propagating the definition mechanically at other types.

Logical relations provide the infrastructure for conducting such proofs.

Contents

Logical equivalence for closed terms

Unary logical relations interpret types by predicates on (*i.e.* sets of) closed values of that type.

Binary relations interpret types by binary relations on closed values of that type, *i.e.* sets of pairs of related values of that type.

That is $\mathcal{V}[\![\tau]\!] \subseteq \mathsf{Val}(\tau) \times \mathsf{Val}(\tau)$.

Then, $\mathcal{E}[\![\tau]\!]$ is the closure of $\mathcal{V}[\![\tau]\!]$ by inverse reduction

We have $\mathcal{V}[\![\tau]\!] \subseteq \mathcal{E}[\![\tau]\!] \subseteq \mathsf{Exp}(\tau) \times \mathsf{Exp}(\tau)$.

Logical equivalence for closed terms

We recursively define two relations $\mathcal{V}[\![\tau]\!]$ and $\mathcal{E}[\![\tau]\!]$ between values of type τ and expressions of type τ by

$$\mathcal{V}[\![\mathsf{B}]\!] \triangleq \{(\mathsf{tt}, \mathsf{tt}), (\mathsf{ff}, \mathsf{ff})\}$$

$$\mathcal{V}[\![\tau \to \sigma]\!] \triangleq \{(V_1, V_2) \mid V_1, V_2 \vdash \tau \to \sigma \wedge \forall (W_1, W_2) \in \mathcal{V}[\![\tau]\!],\ (V_1\ W_1, V_2\ W_2) \in \mathcal{E}[\![\sigma]\!]\}$$

$$\mathcal{E}[\![\tau]\!] \triangleq \{(M_1, M_2) \mid M_1, M_2 : \tau \wedge \exists (V_1, V_2) \in \mathcal{V}[\![\tau]\!],\ M_1 \Downarrow V_1 \wedge M_2 \Downarrow V_2\}$$

where $\Downarrow (M_1, M_2)$ *means* $\{(V_1, V_2) \mid M_i \Downarrow V_i\}$

In the following we will leave the typing constraint in gray implicit (as a global condition for sets $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{E}[\![\cdot]\!]$).

We also write

$M_1 \sim_\tau M_2$ for $(M_1, M_2) \in \mathcal{E}[\![\tau]\!]$ and
$V_1 \approx_\tau V_2$ for $(V_1, V_2) \in \mathcal{V}[\![\tau]\!]$.

Logical equivalence for closed terms (variant)

In a language with non-termination

We change the definition of $\mathcal{E}[\![\tau]\!]$ to

$$\begin{aligned}\mathcal{E}[\![\tau]\!] \;\triangleq\; \Big\{(M_1, M_2) \;\Big|\; & M_1, M_2 : \tau \wedge \\ & \big(\forall V_1,\ M_1 \Downarrow V_1 \implies \exists V_2,\ M_2 \Downarrow V_2 \wedge (V_1, V_2) \in \mathcal{V}[\![\tau]\!]\big) \\ \wedge\, & \big(\forall V_2,\ M_2 \Downarrow V_2 \implies \exists V_1,\ M_1 \Downarrow V_1 \wedge (V_1, V_2) \in \mathcal{V}[\![\tau]\!]\big)\Big\}\end{aligned}$$

Notice

$$\begin{aligned}\mathcal{V}[\![\tau \to \sigma]\!] \;&\triangleq\; \{(V_1, V_2) \mid V_1, V_2 \vdash \tau \to \sigma \wedge \\ &\qquad \forall (W_1, W_2) \in \mathcal{V}[\![\tau]\!],\ (V_1\ W_1, V_2\ W_2) \in \mathcal{E}[\![\sigma]\!]\} \\ &= \{((\lambda x\!:\!\tau.\, M_1), (\lambda x\!:\!\tau.\, M_2)) \mid (\lambda x\!:\!\tau.\, M_1), (\lambda x\!:\!\tau.\, M_2) \vdash \tau \to \sigma \wedge \\ &\qquad \forall (W_1, W_2) \in \mathcal{V}[\![\tau]\!],\ ((\lambda x\!:\!\tau.\, M_1)\ W_1, (\lambda x\!:\!\tau.\, M_2)\ W_2) \in \mathcal{E}[\![\sigma]\!]\}\end{aligned}$$

Properties of logical equivalence for closed terms

Closure by reduction

By definition, since reduction is deterministic: Assume $M_1 \Downarrow N_1$ and $M_2 \Downarrow N_2$ and $(M_1, M_2) \in \mathcal{E}[\![\tau]\!]$, *i.e.* there exists $(V_1, V_2) \in \mathcal{V}[\![\tau]\!]$ $(\mathbf{1})$ such that $M_i \Downarrow V_i$. Since reduction is deterministic, we must have $M_i \Downarrow N_i \Downarrow V_i$. This, together with (1), implies $(N_1, M_2) \in \mathcal{E}[\![\tau]\!]$.

Closure by inverse reduction

Immediate, by construction of $\mathcal{E}[\![\tau]\!]$.

Corollaries

- If $(M_1, M_2) \in \mathcal{E}[\![\tau \to \sigma]\!]$ and $(N_1, N_2) \in \mathcal{E}[\![\tau]\!]$, then $(M_1\ N_1, M_2\ N_2) \in \mathcal{E}[\![\sigma]\!]$.
- To prove $(M_1, M_2) \in \mathcal{E}[\![\tau \to \sigma]\!]$, it suffices to show $(M_1\ V_1, M_2\ V_2) \in \mathcal{E}[\![\sigma]\!]$ for all $(V_1, V_2) \in \mathcal{V}[\![\tau]\!]$.

Properties of logical equivalence for closed terms

Consistency $(\sim_{\mathsf{B}}) \subseteq (\simeq)$

Immediate, by definition of $\mathcal{E}[\![\mathsf{B}]\!]$ and $\mathcal{V}[\![\mathsf{B}]\!] \subseteq (\simeq)$.

Lemma

Logical equivalence is symmetric and transitive (at any given type).

Note: Reflexivity is not at all obvious.

Proof

We show it simultaneously for $\sim_\tau$ and $\approx_\tau$ by induction on type τ.

Logical equivalence for closed terms

We inductively define $M_1 \sim_\tau M_2$ (read M_1 and M_2 are logically equivalent at type τ) on closed terms of (ground) type τ by induction on τ:

- $M_1 \sim_{\mathsf{B}} M_2$ iff $\varnothing \vdash M_1, M_2 : \mathsf{B}$ and $M_1 \simeq M_2$
- $M_1 \sim_{\tau\to\sigma} M_2$ iff $\varnothing \vdash M_1, M_2 : \tau \to \sigma$ and $\forall N_1, N_2,\ \ N_1 \sim_\tau N_2 \implies M_1\, N_1 \sim_\sigma M_2\, N_2$

Lemma

Logical equivalence is symmetric and transitive (at any given type).

Note

Reflexivity is not at all obvious.

Properties of logical equivalence for closed terms (proof)

For $\sim_\tau$, the proof is immediate by transitivity and symmetry of $\approx_\tau$.

For $\approx_\tau$, it goes as follows.

Case τ is B *for values*: the result is immediate.

Case τ is $\tau \to \sigma$:

By IH, symmetry and transitivity hold at types τ and σ.

For symmetry, assume $V_1 \approx_{\tau\to\sigma} V_2$ (H), we must show $V_2 \approx_{\tau\to\sigma} V_1$.

Assume $W_1 \approx_\tau W_2$. We must show $V_2\, W_1 \sim_\sigma V_1\, W_2$ (C). We have $W_2 \approx_\tau W_1$ by symmetry at type τ. By (H), we have $V_2\, W_2 \sim_\sigma V_1\, W_1$ and (C) follows by symmetry of $\sim$ at type σ.

For transitivity, assume $V_1 \approx_{\tau\to\sigma} V_2$ (H1) and $V_2 \approx_{\tau\to\sigma} V_3$ (H2). To show $V_1 \approx_{\tau\to\sigma} V_3$, we assume $W_1 \approx_\tau W_3$ and show $V_1\, W_1 \sim_\sigma V_3\, W_3$ (C).
By (H1), we have $V_1\, W_1 \sim_\sigma V_2\, W_3$ (C1).
By symmetry and transitivity of $\approx_\tau$ (IH), we get $W_3 \approx_\tau W_3$. *It's not reflexivity!*
By (H2), we have $V_2\, W_3 \sim_\sigma V_3\, W_3$ (C2).
(C) follows by transitivity of $\sim_\sigma$ applied to (C1) and (C2).

Logical equivalence for open terms

When $\Gamma \vdash M_1 : \tau$ and $\Gamma \vdash M_2 : \tau$, we wish to define a judgment $\Gamma \vdash M_1 \sim M_2 : \tau$ to mean that the open terms M_1 and M_2 are equivalent at type τ.

The solution is to interpret program variables of $\mathrm{dom}(\Gamma)$ by pairs of related values and typing contexts Γ by a set of (closing) bisubstitutions γ mapping variable type assignments to pairs of related values.

$$\begin{aligned} \mathcal{G}[\![\varnothing]\!] &\triangleq \{\varnothing\} \\ \mathcal{G}[\![\Gamma, x : \tau]\!] &\triangleq \{\gamma, x \mapsto (V_1, V_2) \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \wedge (V_1, V_2) \in \mathcal{V}[\![\tau]\!]\} \end{aligned}$$

Given a bisubstitution γ, we write γ_i for the substitution that maps x to V_i whenever γ maps x to (V_1, V_2).

Definition

$$\Gamma \vdash M_1 \sim M_2 : \tau \iff \forall \gamma \in \mathcal{G}[\![\Gamma]\!],\ (\gamma_1 M_1, \gamma_2 M_2) \in \mathcal{E}[\![\tau]\!]$$

We also write $\vdash M_1 \sim M_2 : \tau$ or $M_1 \sim_\tau M_2$ for $\varnothing \vdash M_1 \sim M_2 : \tau$.

Properties of logical equivalence for open terms

Immediate properties

Open logical equivalence is symmetric and transitive.

(Proof is immediate by the definition and the symmetry and transitivity of closed logical equivalence.)

Fundamental lemma of logical equivalence

Theorem (Reflexivity) *(also called the fundamental lemma))*

If $\Gamma \vdash M : \tau$, *then* $\Gamma \vdash M \sim M : \tau$.

Proof By induction on the typing derivation, using compatibility lemmas.

Compatibility lemmas

C-True
$$\Gamma \vdash \mathsf{tt} \sim \mathsf{tt} : \mathit{bool}$$

C-False
$$\Gamma \vdash \mathsf{ff} \sim \mathsf{ff} : \mathit{bool}$$

C-Var
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \sim x : \tau}$$

C-Abs
$$\frac{\Gamma, x : \tau \vdash M_1 \sim M_2 : \sigma}{\Gamma \vdash \lambda x{:}\tau.\, M_1 \sim \lambda x{:}\tau.\, M_2 : \tau \to \sigma}$$

C-App
$$\frac{\Gamma \vdash M_1 \sim M_2 : \tau \to \sigma \qquad \Gamma \vdash N_1 \sim N_2 : \tau}{\Gamma \vdash M_1\, N_1 \sim M_2\, N_2 : \sigma}$$

C-If
$$\frac{\Gamma \vdash M_1 \sim M_2 : \mathsf{B} \qquad \Gamma \vdash N_1 \sim N_2 : \tau \qquad \Gamma \vdash N_1' \sim N_2' : \tau}{\Gamma \vdash \mathsf{if}\ M_1\ \mathsf{then}\ N_1\ \mathsf{else}\ N_1' \sim \mathsf{if}\ M_2\ \mathsf{then}\ N_2\ \mathsf{else}\ N_2' : \tau}$$

Proof of compatibility lemmas

Each case can be shown independently.

Rule C-ABS: Assume $\Gamma, x : \tau \vdash M_1 \sim M_2 : \sigma$ $(\mathbf{1})$
We show $\Gamma \vdash \lambda x{:}\tau.\, M_1 \sim \lambda x{:}\tau.\, M_2 : \tau \to \sigma$. Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$.
We show $(\gamma_1(\lambda x{:}\tau.\, M_1), \gamma_2(\lambda x{:}\tau.\, M_2)) \in \mathcal{V}[\![\tau \to \sigma]\!]$. Let (V_1, V_2) be in $\mathcal{V}[\![\tau]\!]$.
We show $(\gamma_1(\lambda x{:}\tau.\, M_1)\ V_1, \gamma_2(\lambda x{:}\tau.\, M_2)\ V_2) \in \mathcal{E}[\![\sigma]\!]$ $(\mathbf{2})$.

Since $\gamma_i(\lambda x{:}\tau.\, M_i)\ V_i \Downarrow (\gamma_i, x \mapsto V_i)M_i \triangleq \gamma'_i M_i$, by inverse reduction, it suffices to show $(\gamma'_1 M_1, \gamma'_2 M_2) \in \mathcal{E}[\![\sigma]\!]$. This follows from (1) since $\gamma' \in \mathcal{G}[\![\Gamma, x : \tau]\!]$.

Rule C-APP *(and* C-IF*)*: By induction hypothesis and the fact that substitution distributes over applications (and conditional).

We must show $\Gamma \vdash M_1\ N_1 \sim M_2\ M_2 : \sigma$ $(\mathbf{1})$. Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. From the premises $\Gamma \vdash M_1 \sim M_2 : \tau \to \sigma$ and $\Gamma \vdash N_1 \sim N_2 : \tau$, we have $(\gamma_1 M_1, \gamma_2 M_2) \in \mathcal{E}[\![\tau \to \sigma]\!]$ and $(\gamma_1 N_1, \gamma_2 N_2) \in \mathcal{E}[\![\tau]\!]$. Therefore $(\gamma_1 M_1\ \gamma_1 N_1, \gamma_2 M_2\ \gamma_2 N_2) \in \mathcal{E}[\![\sigma]\!]$. That is $(\gamma_1(M_1\ N_1), \gamma_2(M_2\ N_2)) \in \mathcal{E}[\![\sigma]\!]$, which proves (1).

Rule C-TRUE, C-FALSE, *and* C-VAR: are immediate

Proof of compatibility lemmas (cont.)

Rule C-IF: We show $\Gamma \vdash$ if M_1 then N_1 else $N_1' \sim$ if M_2 then N_2 else $N_2' : \tau$.
Assume $\gamma \in \mathcal{G}[\![\gamma]\!]$.
We show $(\gamma_1(\text{if } M_1 \text{ then } N_1 \text{ else } N_1'), \gamma_2(\text{if } M_2 \text{ then } N_2 \text{ else } N_2')) \in \mathcal{E}[\![\tau]\!]$, That is (if $\gamma_1 M_1$ then $\gamma_1 N_1$ else $\gamma_1 N_1'$, if $\gamma_2 M_2$ then $\gamma_2 N_2$ else $\gamma_2 N_2'$) $\in \mathcal{E}[\![\tau]\!]$ $(\mathbf{1})$.

From the premise $\Gamma \vdash M_1 \sim M_2 : \mathsf{B}$, we have $(\gamma_1 M_1, \gamma_2 M_2) \in \mathcal{E}[\![\mathsf{B}]\!]$. Therefore $M_1 \Downarrow V$ and $M_2 \Downarrow V$ where V is either tt or ff:

- *Case V is* tt:. Then, (if $\gamma_i M_i$ then $\gamma_i N_i$ else $\gamma_i N_i'$) $\Downarrow \gamma_i N_i$, *i.e.* γ_i(if M_i then N_i else N_i') $\Downarrow \gamma_i N_i$. From the premise $\Gamma \vdash N_1 \sim N_2 : \tau$, we have $(\gamma_1 N_1, \gamma_2 N_2) \in \mathcal{E}[\![\tau]\!]$ and (1) follows by closer by inverse reduction.
- *Case V is* ff: similar.

Proof of reflexivity

By induction on the derivation of $\Gamma \vdash M : \tau$.
We must show $\Gamma \vdash M \sim M : \tau$:

All cases immediately follow from compatibility lemmas.

Case M *is* tt *or* ff: Immediate by Rule C-True or Rule C-False

Case M *is* x: Immediate by Rule C-Var.

Case M *is* $M' \; N$: By inversion of the typing rule App, induction hypothesis, and Rule C-App.

Case M *is* $\lambda \tau : N.$: By inversion of the typing rule Abs, induction hypothesis, and Rule C-Abs.

Properties of logical relations

Corollary (equivalence) Open logical relation is an equivalence relation

Logical equivalence is a congruence
If $\Gamma \vdash M \sim M' : \tau$ and $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Delta \rhd \sigma)$, then
$\Delta \vdash \mathcal{C}[M] \sim \mathcal{C}[M'] : \sigma$.

Proof By induction on the proof of $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Delta \rhd \sigma)$.

Similar to the proof of reflexivity—but *we need a syntactic definition of context-typing derivations* (which we have omitted) to be able to reason by induction on the context-typing derivation.

Soundness of logical equivalence
Logical equivalence implies observational equivalence.
If $\Gamma \vdash M \sim M' : \tau$ then $\Gamma \vdash M \cong M' : \tau$.

Proof: Logical equivalence is a consistent congruence, hence included in observational equivalence which is the coarsiest such relation.

Properties of logical equivalence

Completeness of logical equivalence
Observational equivalence of closed terms implies logical equivalence.
That is $(\cong_\tau) \subseteq (\sim_\tau)$.

Proof by induction on τ.

Case B: In the empty context, by consistency $\cong_\mathsf{B}$ is a subrelation of $\simeq_\mathsf{B}$ which coincides with $\sim_\mathsf{B}$.

Case $\tau \to \sigma$: By congruence of observational equivalence!

By hypothesis, we have $M_1 \cong_{\tau\to\sigma} M_2$ $(\mathbf{1})$. To show $M_1 \sim_{\tau\to\sigma} M_2$, we assume $V_1 \approx_\tau V_2$ $(\mathbf{2})$ and show $M_1\, V_1 \sim_\sigma M_2\, V_2$ $(\mathbf{3})$.

By soundness applied to (2), we have $V_1 \cong_\tau V_2$ from (2). By congruence with (1), we have $M_1\, V_1 \cong_\sigma M_2\, V_2$, which implies (3) by IH at type σ.

Logical equivalence: example of application

Fact: Assume $not \triangleq \lambda x{:}\mathsf{B}.\,\mathsf{if}\ x\ \mathsf{then\ ff\ else\ tt}$
and $M \triangleq \lambda x{:}\mathsf{B}.\,\lambda y{:}\tau.\,\lambda z{:}\tau.\,\mathsf{if}\ not\ x\ \mathsf{then}\ y\ \mathsf{else}\ z$
and $M' \triangleq \lambda x{:}\mathsf{B}.\,\lambda y{:}\tau.\,\lambda z{:}\tau.\,\mathsf{if}\ x\ \mathsf{then}\ z\ \mathsf{else}\ y$.

Show that $M \cong_{\mathsf{B}\to\tau\to\tau\to\tau} M'$.

Proof

It suffices to show $M\ V_0\ V_1\ V_2 \sim_\tau M'\ V_0'\ V_1'\ V_2'$ whenever $V_0 \approx_{\mathsf{B}} V_0'$ $(\mathbf{1})$ and $V_1 \approx_\tau V_1'$ $(\mathbf{2})$ and $V_2 \approx_\tau V_2'$ $(\mathbf{3})$. By inverse reduction, it suffices to show: $\mathsf{if}\ not\ V_0\ \mathsf{then}\ V_1\ \mathsf{else}\ V_2 \ \sim_\tau\ \mathsf{if}\ V_0'\ \mathsf{then}\ V_2'\ \mathsf{else}\ V_1'$ $(\mathbf{4})$.

It follows from (1) that we have only two cases:

Case $V_0 = V_0' = \mathsf{tt}$: Then $not\ V_0 \Downarrow \mathsf{ff}$ and thus $M \Downarrow V_2$ while $M' \Downarrow V_2$. Then (4) follows by inverse reduction and (3).

Case $V_0 = V_0' = \mathsf{ff}$: is symmetric.

Contents

Observational equivalence

We now extend the notion of logical equivalence to System F.

$$\tau ::= \ldots \mid \alpha \mid \forall\alpha.\,\tau \qquad\qquad M ::= \ldots \mid \Lambda\alpha.\,M \mid M\ \tau$$

We write typing contexts $\Delta;\Gamma$ where Δ binds variables and Γ binds program variables.

Typing of contexts becomes $\mathcal{C} : (\Delta;\Gamma \rhd \tau) \rightsquigarrow (\Delta';\Gamma' \rhd \tau')$.

Observational equivalence

We (re)defined $\Delta;\Gamma \vdash M \cong M' : \tau$ as

$$\forall \mathcal{C} : (\Delta;\Gamma \rhd \tau) \rightsquigarrow (\varnothing;\varnothing \rhd \mathsf{B}),\ \ \mathcal{C}[M] \simeq \mathcal{C}[M']$$

As before, write $M \cong_\tau N$ for $\varnothing;\varnothing \vdash M \cong N : \tau$ (in particular, τ is closed).

Logical equivalence

For closed terms (no free program variables)

- We need to give the semantics of polymoprhic types $\forall \alpha. \tau$
- Problem: We cannot do it in terms of the semantics of instances $\tau[\alpha \mapsto \sigma]$ since the semantics is defined by induction on types.
- Solution: we give the semantics of terms with open types—in some suitable environment that interprets type variables by logical relations (sets of pairs of related values) of closed types ρ_1 and ρ_2

Let $\mathcal{R}(\rho_1, \rho_2)$ be the set of relations on values of closed types ρ_1 and ρ_2, that is $\mathcal{P}(\mathsf{Val}(\rho_1) \times \mathsf{Val}(\rho_2))$. We optionally restrict to *admissible* relations, *i.e.* relations that are *closed by observational equivalence*:

$$R \in \mathcal{R}^\sharp(\tau_1, \tau_2) \implies \forall (V_1, V_2) \in R,\ \forall W_1, W_2,\ W_1 \cong V_1 \wedge W_2 \cong V_2 \implies (W_1, W_2) \in R$$

The restriction to *admissible relations* is required for *completeness* of logical equivalence with respect to observational equivalence but *not for soundness*.

Example of admissible relations

For example, both

$$R_1 \triangleq \{(\mathsf{tt}, 0), (\mathsf{ff}, 1)\}$$
$$R_2 \triangleq \{(\mathsf{tt}, 0)\} \cup \{(\mathsf{ff}, n) \mid n \in \mathbb{Z}^\star\}$$

are admissible relations in $\mathcal{R}^\sharp(\mathsf{B}, \mathit{int})$.

But

$$R_3 \triangleq \{(\mathsf{tt}, \lambda x\!:\!\tau.\, 0), (\mathsf{ff}, \lambda x\!:\!\tau.\, 1)\}$$

although in $\mathcal{R}(\mathsf{B}, \tau \to \mathit{int})$, is not admissible.

Taking $M_0 \triangleq \lambda x\!:\!\tau.\, (\lambda z\!:\!\mathit{int}.\, z)\; 0$, we have $M \cong_{\tau\to\mathit{int}} \lambda x\!:\!\tau.\, 0$ but (tt, M) is not in R_3. **Note** A relation R in $\mathcal{R}(\tau_1, \tau_2)$ can always be turned into an admissible relation $R^\sharp$ in $\mathcal{R}^\sharp(\tau_1, \tau_2)$ by closing R by observational equivalence.

Note It is *a key* that such relations can relate values at different types.

Interpretation of type environments

Interpretation of type variables

We write η for mappings $\alpha \mapsto (\rho_1, \rho_2, R)$ where $R \in \mathcal{R}(\rho_1, \rho_2)$.

We write η_i (*resp.* η_R) for the type (*resp.* relational) substitution that maps α to ρ_i (*resp.* R) whenever η maps α to (ρ_1, ρ_2, R).

We define

$$\mathcal{V}[\![\alpha]\!]_\eta \triangleq \eta_R(\alpha)$$

$$\begin{aligned}\mathcal{V}[\![\forall\alpha.\,\tau]\!]_\eta \triangleq \{(V_1, V_2) \mid\ & V_1 : \eta_1(\forall\alpha.\,\tau) \wedge V_2 : \eta_2(\forall\alpha.\,\tau) \wedge \\ & \forall \rho_1, \rho_2, \forall R \in \mathcal{R}(\rho_1, \rho_2), (V_1\ \rho_1, V_2\ \rho_2) \in \mathcal{E}[\![\tau]\!]_{\eta, \alpha \mapsto (\rho_1, \rho_2, R)}\}\end{aligned}$$

Logical equivalence for closed terms with open types

We redefine

$$\mathcal{V}[\![\mathsf{B}]\!]_{\eta} \triangleq \{(\mathsf{tt}, \mathsf{tt}), (\mathsf{ff}, \mathsf{ff})\}$$

$$\mathcal{V}[\![\tau \to \sigma]\!]_{\eta} \triangleq \{(V_1, V_2) \mid V_1 \vdash \eta_1(\tau \to \sigma) \wedge V_2 \vdash \eta_2(\tau \to \sigma) \wedge \forall (W_1, W_2) \in \mathcal{V}[\![\tau]\!]_{\eta},\ (V_1\ W_1, V_2\ W_2) \in \mathcal{E}[\![\sigma]\!]_{\eta}\}$$

$$\mathcal{E}[\![\tau]\!]_{\eta} \triangleq \{(M_1, M_2) \mid M_1 : \eta_1\tau \wedge M_2 : \eta_2\tau \wedge \exists (V_1, V_2) \in \mathcal{V}[\![\tau]\!]_{\eta}, M_1 \Downarrow V_1 \wedge M_2 \Downarrow V_2\}$$

$$\mathcal{G}[\![\varnothing]\!]_{\eta} \triangleq \{\varnothing\}$$

$$\mathcal{G}[\![\Gamma, x : \tau]\!]_{\eta} \triangleq \{\gamma, x \mapsto (V_1, V_2) \mid \gamma \in \mathcal{G}[\![\Gamma]\!]_{\eta} \wedge (V_1, V_2) \in \mathcal{V}[\![\tau]\!]_{\eta}\}$$

and define

$$\mathcal{D}[\![\varnothing]\!] \triangleq \{\varnothing\}$$

$$\mathcal{D}[\![\Delta, \alpha]\!] \triangleq \{\eta, \alpha \mapsto (\rho_1, \rho_2, \mathcal{R}) \mid \eta \in \mathcal{D}[\![\Delta]\!] \wedge R \in \mathcal{R}(\rho_1, \rho_2)\}$$

Logical equivalence for open terms

Definition We define $\Delta; \Gamma \vdash M \sim M' : \tau$ as

$$\wedge \begin{cases} \Delta; \Gamma \vdash M, M' : \tau \\ \forall \eta \in \mathcal{D}[\![\Delta]\!],\ \forall \gamma \in \mathcal{G}[\![\Gamma]\!]_\eta,\ (\eta_1(\gamma_1 M_1), \eta_2(\gamma_2 M_2)) \in \mathcal{E}[\![\tau]\!]_\eta \end{cases}$$

(Notations are a bit heavy, but intuitions should remain simple.)

Notation

We also write $M_1 \sim_\tau M_2$ for $\vdash M_1 \sim M_2 : \tau$ (*i.e.* $\varnothing; \varnothing \vdash M_1 \sim M_2 : \tau$).

In this case, τ is a closed type and M_1 and M_2 are closed terms of type τ; hence, this coincides with the previous definition (M_1, M_2) in $\mathcal{E}[\![\tau]\!]$, which may still be used as a shorthand for $\mathcal{E}[\![\tau]\!]_\varnothing$.

Properties

Respect for observational equivalence

If $(M_1, M_2) \in \mathcal{E}[\![\tau]\!]^\sharp_\eta$ and $N_1 \cong_{\eta_1(\tau)} M_1$ and $N_2 \cong_{\eta_2(\tau)} M_2$ then $(N_1, N_2) \in \mathcal{E}[\![\tau]\!]^\sharp_\eta$.
Requires admissibility

(We use $^\sharp$ to indicate that admissibility is required in the definition of $\mathcal{R}^\sharp$)

Proof. By induction on τ.

Assume $(M_1, M_2) \in \mathcal{E}[\![\tau]\!]_\eta$ $(\mathbf{1})$ and $N_1 \cong_{\eta_1(\tau)} M_1$ $(\mathbf{2})$. We show $(N_1, M_2) \in \mathcal{E}[\![\tau]\!]_\eta$.

Case τ is $\forall\alpha.\sigma$: Assume $R \in \mathcal{R}^\sharp(\rho_1, \rho_2)$. Let η_α be $\eta, \alpha \mapsto (\rho_1, \rho_2, R)$.
We have $(M_1\ \rho_1, M_2\ \rho_2) \in \mathcal{E}[\![\sigma]\!]_{\eta_\alpha}$, from (1).
By congruence from (2), we have $N_1 \rho_1 \cong_{\delta(\tau)} M_1\ \rho_1$.
Hence, by induction hypothesis, $(M_1\ \rho_1, M_2\ \rho_2) \in \mathcal{E}[\![\sigma]\!]_{\eta_\alpha}$, as expected.

Case τ is α: Relies on admissibility, indeed.

Other cases: the proof is similar to the case of the simply-typed λ-calculus.

Corollary

Properties

Lemma (Closure under observational equivalence)
If $\Delta;\Gamma \vdash M_1 \sim^\sharp M_2 : \tau$ and $\Delta;\Gamma \vdash M_1 \cong N_1 : \tau$ and $\Delta;\Gamma \vdash M_2 \cong N_2 : \tau$, then $\Delta;\Gamma \vdash N_1 \sim^\sharp N_2 : \tau$

Requires admissibility

Lemma (Compositionality)

Key lemma

Assume $\Delta \vdash \sigma$ and $\Delta, \alpha \vdash \tau$ and $\eta \in \mathcal{D}[\![\Delta]\!]$. Then,

$$\mathcal{V}[\![\tau[\alpha \mapsto \sigma]]\!]_\eta \quad = \quad \mathcal{V}[\![\tau]\!]_{\eta, \alpha \mapsto (\eta_1\sigma, \eta_2\sigma, \mathcal{V}[\![\sigma]\!]_\eta)}$$

Proof by induction on τ.

Parametricity

Theorem (Reflexivity) *(also called the fundamental lemma)*

If $\Delta; \Gamma \vdash M : \tau$ *then* $\Delta; \Gamma \vdash M \sim M : \tau$.

Notice: Admissibility is not required for the fundamental lemma

Proof by induction on the typing derivation, using compatibility lemmas.

Compatibility lemmas

We redefine the lemmas to work in a typing context of the form Δ, Γ instead of Γ and add two new lemmas:

C-TABS
$$\frac{\Delta, \alpha; \Gamma \vdash M_1 \sim M_2 : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.\, M_1 \sim \Lambda\alpha.\, M_2 : \forall\alpha.\, \tau}$$

C-TAPP
$$\frac{\Delta; \Gamma \vdash M_1 \sim M_2 : \forall\alpha.\, \tau \qquad \Delta \vdash \sigma}{\Delta; \Gamma \vdash M_1\, \sigma \sim M_2\, \sigma : \tau[\alpha \mapsto \sigma]}$$

Proof of compatibility

Case M is $\Lambda\alpha.\,N$: We must show that $\Delta;\Gamma \vdash \Lambda\alpha.\,N \sim \Lambda\alpha.\,N : \forall\alpha.\,\tau$.
Assume $\eta : \delta \leftrightarrow_\Delta \delta'$ and $\gamma \sim_\Gamma \gamma' \;\; [\eta : \delta \leftrightarrow \delta']$.

We must show $\gamma(\delta(\Lambda\alpha.\,N)) \sim_{\forall\alpha.\,\tau} \gamma'(\delta(\Lambda\alpha.\,N)) \;\; [\eta : \delta \leftrightarrow \delta]$.

Assume σ and σ' closed and $R : \sigma \leftrightarrow \sigma'$. We must show

$$(\gamma(\delta(\Lambda\alpha.\,N)))\;\sigma \sim_\tau (\gamma'(\delta'(\Lambda\alpha.\,N)))\;\sigma \;\; [\eta_0 : \delta_0 \leftrightarrow \delta'_0]$$

where $\eta_0 = \eta, \alpha \mapsto R$ and $\delta_0 = \delta, \alpha \mapsto \sigma$ and $\delta'_0 = \delta, \alpha \mapsto \sigma'$.
Since

$$(\gamma(\delta(\Lambda\alpha.\,N)))\;\sigma = (\Lambda\alpha.\,\gamma(\delta(N)))\;\sigma \longrightarrow \gamma(\delta(N))[\alpha \mapsto \sigma] = \gamma(\delta_0(N))$$

It suffices to show

$$\gamma(\delta_0(N)) \sim_\tau \gamma'(\delta'_0(N)) \;\; [\eta_0 : \delta_0 \leftrightarrow \delta'_0]$$

which follows by IH from $\Delta, \alpha; \Gamma \vdash N : \tau$ (which we obtain from $\Delta, \Gamma \vdash \Lambda\alpha.\,N : \tau$ by inversion).

Proof of compatibility

Case M is $N\ \sigma$:

By inversion of typing $\Delta, \Gamma \vdash N : \forall \alpha.\, \tau_0$ $(\mathbf{1})$ and τ is $\forall \alpha.\, \tau_0$.
We must show that $\Delta; \Gamma \vdash N\ \sigma \sim N\ \sigma : \tau_0[\alpha \mapsto \sigma]$.

Assume $\eta : \delta \leftrightarrow_\Delta \delta'$ and $\gamma \sim_\Gamma \gamma'\ \ [\eta : \delta \leftrightarrow \delta']$. We must show

$$\gamma(\delta(N\ \sigma)) \sim_{\tau_0[\alpha \mapsto \sigma]} \gamma'(\delta'(N\ \sigma))\ \ [\eta : \delta \leftrightarrow \delta']$$

i.e. $$(\gamma(\delta(N)))\ \sigma \sim_{\tau_0[\alpha \mapsto \sigma]} (\gamma'(\delta'(N)))\ \sigma\ \ [\eta : \delta \leftrightarrow \delta']$$

By compositionality, it suffices to show

$$(\gamma(\delta(N)))\ \sigma \sim_{\tau_0} (\gamma'(\delta'(N)))\ \sigma\ \ [\eta_0 : \delta_0 \leftrightarrow \delta'_0] \qquad (\mathbf{2})$$

where $\eta_0 = \eta, \alpha \mapsto R$ and $\delta_0 = \delta, \alpha \mapsto \sigma$ and $\delta'_0 = \delta, \alpha \mapsto \sigma'$ and
$R : \delta(s) \leftrightarrow \delta'(s)$ is defined by $R(N_0, N'_0) \iff N_0 \sim_\sigma N'_0\ [\eta : \delta \leftrightarrow \delta']$.
This relation is admissible $(\mathbf{3})$. Hence by IH from (1), we have

$$(\gamma(\delta(N)))\ \ \sim_{\forall \alpha.\, \tau_0} (\gamma'(\delta'(N)))\ \ [\eta : \delta \leftrightarrow \delta']$$

which implies (2) by definition of $\sim_{\forall \alpha.\, \tau_0}$.

Properties

Soundness of logical equivalence

Logical equivalence implies observational equivalence.
If $\Delta; \Gamma \vdash M_1 \sim M_2 : \tau$ then $\Delta; \Gamma \vdash M_1 \cong M_2 : \tau$.

Completeness of logical equivalence

Observational equivalence implies logical equivalence with admissibility.
If $\Delta; \Gamma \vdash M_1 \cong M_2 : \tau$ then $\Delta; \Gamma \vdash M_1 \sim^{\sharp} M_2 : \tau$.

As a particular case, $M_1 \cong_\tau M_2$ iff $M_1 \sim^{\sharp}_\tau M_2$.

Note: Admissibility is not required for soundness—only for completeness.

That is, proofs that some observational equivalence hold do not usually require admissibility.

Properties

Extensionality *(A fact, hence does not depend on admissibility)*

$M_1 \cong_{\tau\to\sigma} M_2$ iff $\forall (V : \tau), M_1\ V \cong_\sigma M_2\ V$ iff $\forall (N : \tau), M_1\ N \cong_\sigma M_2\ N$

$M_1 \cong_{\forall\alpha.\,\tau} M_2$ iff for all closed type ρ, $M_1\ \rho \cong_{\tau[\alpha\mapsto\rho]} M_2\ \rho$.

Proof. Forward direction is immediate as $\cong$ is a congruence. Backward direction uses logical relations and admissibility, but the exported statement does not.

Case Value abstraction: It suffices to show $M_1 \sim_{\tau\to\sigma} M_2$. That is, assuming $N_1 \sim_\tau N_2$ $(\mathbf{1})$, we show $M_1\ N_1 \sim_\sigma M_2\ N_2$ $(\mathbf{2})$. By assumption, we have $M_1\ N_1 \cong_\sigma M_2\ N_1$ $(\mathbf{3})$. By the fundamental lemma, we have $M_2 \sim_{\tau\to\sigma} M_2$. Hence, from (1), we must have $M_2\ N_1 \sim_\sigma M_2\ N_2$, We conclude (2) by *respect for observational equivalence* with (3)—which requires admissibility.

Case Type abstraction: It suffices to show $M_1 \sim_{\forall\alpha.\,\tau} M_2$. That is, given $R \in \mathcal{R}(\rho_1, \rho_2)$, we show $(M_1\ \rho_1, M_2\ \rho_2) \in \mathcal{E}[\![\tau]\!]_{\alpha\mapsto(\rho_1,\rho_2,R)}$ $(\mathbf{4})$.
By assumption, we have $M_1\ \rho_1 \cong_{\tau[\alpha\mapsto\rho_1]} M_2\ \rho_1$ $(\mathbf{5})$.
By the fundamental lemma, we have $M_2 \sim_{\forall\alpha.\,\tau} M_2$.
Hence, we have $(M_2\ \rho_1, M_2\ \rho_2) \in \mathcal{E}[\![\tau]\!]_{\alpha\mapsto(\rho_1,\rho_2,R)}$
We conclude (4) by *respect for observational equivalence* with (5).

Properties

Identity extension Requires admissibiily

Let θ be a substitution of type variables for ground types.
Let R be the restriction of $\cong_{\alpha\theta}$ to $\mathsf{Val}(\alpha\theta) \times \mathsf{Val}(\alpha\theta))$ and $\eta : \alpha \mapsto (\alpha\theta, \alpha\theta, R)$.

Then $\mathcal{E}[\![\tau]\!]_\eta$ is equal to $\cong_{\tau\theta}$.

(The proof uses respect for observational equivalence, which requires admissibility)

Contents

Applications — Inhabitants of $\forall\alpha.\,\alpha \to \alpha$

Fact If $M : \forall\alpha.\,\alpha \to \alpha$, then $M \cong_{\forall\alpha.\,\alpha\to\alpha} id$ where $id \triangleq \Lambda\alpha.\,\lambda x{:}\alpha.\,x$.

Proof By *extensionality*, it suffices to show that for any ρ and $V : \rho$ we have $M\ \rho\ V \cong_\rho id\ \rho\ V$. In fact, by closure by inverse reduction, it suffices to show $M\ \rho\ V \cong_\rho V$ $(\mathbf{1})$.

By parametricity, we have $M \sim_{\forall\alpha.\,\alpha\to\alpha} M$ $(\mathbf{2})$.

Consider R in $\mathcal{R}(\rho,\rho)$ equal to $\{(V,V)\}$ and η be $[\alpha \mapsto (\rho,\rho,R)]$. (3)
By construction, we have $(V,V) \in \mathcal{V}[\![\alpha]\!]_\eta$.

Hence, from (2), we have $(M\ \rho\ V, M\ \rho\ V) \in \mathcal{E}[\![\alpha]\!]_\eta$, which means that the pair $(M\ \rho\ V, M\ \rho\ V)$ reduces to a pair of values in (the singleton) R. This implies that $M\ \rho\ V$ reduces to V, which in turn, implies (1).

$(\mathbf{3})$ Admissibility is not needed

Applications

Inhabitants of $\forall\alpha.\,\alpha \to \alpha \to \alpha$

Fact Let σ be $\forall\alpha.\,\alpha \to \alpha \to \alpha$. If $M : \sigma$, then either
$M \cong_\sigma W_1 \triangleq \Lambda\alpha.\,\lambda x_1{:}\alpha.\,\lambda x_2{:}\alpha.\,x_1$ or $M \cong_\sigma W_2 \triangleq \Lambda\alpha.\,\lambda x_1{:}\alpha.\,\lambda x_2{:}\alpha.\,x_2$

Proof By *extensionality*, it suffices to show that for either $i = 1$ or $i = 2$, for any closed type ρ and $V_1, V_2 : \rho$, we have $M\ \rho\, V_1\ V_2 \cong_\rho W_i\ \rho\, V_1\ V_2$, or just $M\ \rho\, V_1\ V_2 \cong_\sigma V_i$ (**1**).

Let ρ and $V_1, V_2 : \rho$ be fixed. Consider R equal to $\{(\mathsf{tt}, V_1), (\mathsf{ff}, V_2)\}$ in $\mathcal{R}(\mathsf{B}, \rho)$ and η be $\alpha \mapsto (\mathsf{B}, \rho, R)$. We have $(\mathsf{tt}, V_1) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(\mathsf{tt}, V_1)$ and, similarly, $(\mathsf{ff}, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.

We have $(M, M) \in \mathcal{E}[\![\sigma]\!]$ by parametricity. Hence, $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\, V_1\ V_2)$ is in $\mathcal{V}[\![\alpha]\!]_\eta$, which means that $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\, V_1\ V_2)$ reduces to a pair of values in R, which implies:

$$\forall \rho, V_1, V_2, \quad \bigvee \begin{cases} \forall \rho, V_1, V_2,\ \ M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{tt} & \wedge \quad M\ \rho\, V_1\ V_2 \cong_\rho V_1 \\ \forall \rho, V_1, V_2,\ \ M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{ff} & \wedge \quad M\ \rho\, V_1\ V_2 \cong_\rho V_2 \end{cases}$$

Since, $M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}$ is independent of ρ, V_1, and V_2, this actually shows (1).

Applications

Inhabitants of $\forall\alpha.\,\alpha \to \alpha \to \alpha$

Fact Let σ be $\forall\alpha.\,\alpha \to \alpha \to \alpha$. If $M : \sigma$, then either
$M \cong_\sigma W_1 \triangleq \Lambda\alpha.\,\lambda x_1{:}\alpha.\,\lambda x_2{:}\alpha.\,x_1$ or $M \cong_\sigma W_2 \triangleq \Lambda\alpha.\,\lambda x_1{:}\alpha.\,\lambda x_2{:}\alpha.\,x_2$

Proof By *extensionality*, it suffices to show that for either $i = 1$ or $i = 2$, for any closed type ρ and $V_1, V_2 : \rho$, we have $M\ \rho\ V_1\ V_2 \cong_\rho W_i\ \rho\ V_1\ V_2$, or just $M\ \rho\ V_1\ V_2 \cong_\sigma V_i$ (**1**).

Let ρ and $V_1, V_2 : \rho$ be fixed. Consider R equal to $\{(\mathsf{tt}, V_1), (\mathsf{ff}, V_2)\}$ in $\mathcal{R}(\mathsf{B}, \rho)$ and η be $\alpha \mapsto (\mathsf{B}, \rho, R)$. We have $(\mathsf{tt}, V_1) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(\mathsf{tt}, V_1)$ and, similarly, $(\mathsf{ff}, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.

We have $(M, M) \in \mathcal{E}[\![\sigma]\!]$ by parametricity. Hence, $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\ V_1\ V_2)$ is in $\mathcal{V}[\![\alpha]\!]_\eta$, which means that $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\ V_1\ V_2)$ reduces to a pair of values in R, which implies:

$$\forall\rho, V_1, V_2, \quad \bigvee \begin{cases} \forall\rho, V_1, V_2, \ \ M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{tt} \ \wedge\ M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ \forall\rho, V_1, V_2, \ \ M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{ff} \ \wedge\ M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

Since, $M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}$ is independent of ρ, V_1, and V_2, this actually shows (1).

Applications

Inhabitants of $\forall\alpha.\,\alpha \to \alpha \to \alpha$

Fact Let σ be $\forall\alpha.\,\alpha \to \alpha \to \alpha$. If $M : \sigma$, then either $M \cong_\sigma W_1 \triangleq \Lambda\alpha.\,\lambda x_1{:}\alpha.\,\lambda x_2{:}\alpha.\,x_1$ or $M \cong_\sigma W_2 \triangleq \Lambda\alpha.\,\lambda x_1{:}\alpha.\,\lambda x_2{:}\alpha.\,x_2$

Proof By *extensionality*, it suffices to show that for either $i = 1$ or $i = 2$, for any closed type ρ and $V_1, V_2 : \rho$, we have $M\ \rho\ V_1\ V_2 \cong_\rho W_i\ \rho\ V_1\ V_2$, or just $M\ \rho\ V_1\ V_2 \cong_\sigma V_i$ (**1**).

Let ρ and $V_1, V_2 : \rho$ be fixed. Consider R equal to $\{(\mathsf{tt}, V_1), (\mathsf{ff}, V_2)\}$ in $\mathcal{R}(\mathsf{B}, \rho)$ and η be $\alpha \mapsto (\mathsf{B}, \rho, R)$. We have $(\mathsf{tt}, V_1) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(\mathsf{tt}, V_1)$ and, similarly, $(\mathsf{ff}, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.

We have $(M, M) \in \mathcal{E}[\![\sigma]\!]$ by parametricity. Hence, $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\ V_1\ V_2)$ is in $\mathcal{V}[\![\alpha]\!]_\eta$, which means that $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\ V_1\ V_2)$ reduces to a pair of values in R, which implies:

$$\forall \rho, V_1, V_2, \quad \bigvee \begin{cases} \forall \rho, V_1, V_2, & M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{tt} \ \wedge\ M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ \forall \rho, V_1, V_2, & M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{ff} \ \wedge\ M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

Since, $M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}$ is independent of ρ, V_1, and V_2, this actually shows (1).

Applications

Inhabitants of $\forall \alpha. \alpha \to \alpha \to \alpha$

Fact Let σ be $\forall \alpha. \alpha \to \alpha \to \alpha$. If $M : \sigma$, then either
$M \cong_\sigma W_1 \triangleq \Lambda \alpha. \lambda x_1 {:} \alpha. \lambda x_2 {:} \alpha. x_1$ or $M \cong_\sigma W_2 \triangleq \Lambda \alpha. \lambda x_1 {:} \alpha. \lambda x_2 {:} \alpha. x_2$

Proof By *extensionality*, it suffices to show that for either $i = 1$ or $i = 2$, for any closed type ρ and $V_1, V_2 : \rho$, we have $M\ \rho\ V_1\ V_2 \cong_\rho W_i\ \rho\ V_1\ V_2$, or just $M\ \rho\ V_1\ V_2 \cong_\sigma V_i$ (**1**).

Let ρ and $V_1, V_2 : \rho$ be fixed. Consider R equal to $\{(\text{tt}, V_1), (\text{ff}, V_2)\}$ in $\mathcal{R}(\text{B}, \rho)$ and η be $\alpha \mapsto (\text{B}, \rho, R)$. We have $(\text{tt}, V_1) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(\text{tt}, V_1)$ and, similarly, $(\text{ff}, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.

We have $(M, M) \in \mathcal{E}[\![\sigma]\!]$ by parametricity. Hence, $(M\ \text{B}\ \text{tt}\ \text{ff}, M\ \rho\ V_1\ V_2)$ is in $\mathcal{V}[\![\alpha]\!]_\eta$, which means that $(M\ \text{B}\ \text{tt}\ \text{ff}, M\ \rho\ V_1\ V_2)$ reduces to a pair of values in R, which implies:

$$\forall \rho, V_1, V_2, \quad \bigvee \begin{cases} \forall \rho, V_1, V_2, \quad M\ \text{B}\ \text{tt}\ \text{ff} \cong_{\text{B}} \text{tt} \ \wedge\ M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ \forall \rho, V_1, V_2, \quad M\ \text{B}\ \text{tt}\ \text{ff} \cong_{\text{B}} \text{ff} \ \wedge\ M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

Since, M B tt ff is independent of ρ, V_1, and V_2, this actually shows (1).

Applications

Inhabitants of $\forall\alpha.\,\alpha \to \alpha \to \alpha$

Fact Let σ be $\forall\alpha.\,\alpha \to \alpha \to \alpha$. If $M : \sigma$, then either
$M \cong_\sigma W_1 \triangleq \Lambda\alpha.\,\lambda x_1{:}\alpha.\,\lambda x_2{:}\alpha.\,x_1$ or $M \cong_\sigma W_2 \triangleq \Lambda\alpha.\,\lambda x_1{:}\alpha.\,\lambda x_2{:}\alpha.\,x_2$

Proof By *extensionality*, it suffices to show that for either $i = 1$ or $i = 2$, for any closed type ρ and $V_1, V_2 : \rho$, we have $M\ \rho\ V_1\ V_2 \cong_\rho W_i\ \rho\ V_1\ V_2$, or just $M\ \rho\ V_1\ V_2 \cong_\sigma V_i$ (**1**).

Let ρ and $V_1, V_2 : \rho$ be fixed. Consider R equal to $\{(\mathbf{0}, V_1), (\mathbf{1}, V_2)\}$ in $\mathcal{R}(\mathbb{N}, \rho)$ and η be $\alpha \mapsto (\mathbb{N}, \rho, R)$. We have $(\mathbf{0}, V_1) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(\mathbf{0}, V_1)$ and, similarly, $(\mathbf{1}, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.

We have $(M, M) \in \mathcal{E}[\![\sigma]\!]$ by parametricity. Hence, $(M\ \mathbb{N}\ \mathbf{0}\ \mathbf{1}, M\ \rho\ V_1\ V_2)$ is in $\mathcal{V}[\![\alpha]\!]_\eta$, which means that $(M\ \mathbb{N}\ \mathbf{0}\ \mathbf{1}, M\ \rho\ V_1\ V_2)$ reduces to a pair of values in R, which implies:

$$\forall \rho, V_1, V_2, \quad \bigvee \begin{cases} \forall \rho, V_1, V_2, & M\ \mathbb{N}\ \mathbf{0}\ \mathbf{1} \cong_{\mathbb{N}} \mathbf{0} \quad \wedge \quad M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ \forall \rho, V_1, V_2, & M\ \mathbb{N}\ \mathbf{0}\ \mathbf{1} \cong_{\mathbb{N}} \mathbf{1} \quad \wedge \quad M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

Since, $M\ \mathbb{N}\ \mathbf{0}\ \mathbf{1}$ is independent of ρ, V_1, and V_2, this actually shows (1).

Applications

Inhabitants of $\forall\alpha.\,\alpha \to \alpha \to \alpha$

Fact Let σ be $\forall\alpha.\,\alpha \to \alpha \to \alpha$. If $M : \sigma$, then either
$M \cong_\sigma W_1 \triangleq \Lambda\alpha.\,\lambda x_1{:}\alpha.\,\lambda x_2{:}\alpha.\,x_1$ or $M \cong_\sigma W_2 \triangleq \Lambda\alpha.\,\lambda x_1{:}\alpha.\,\lambda x_2{:}\alpha.\,x_2$

Proof By *extensionality*, it suffices to show that for either $i = 1$ or $i = 2$, for any closed type ρ and $V_1, V_2 : \rho$, we have $M\ \rho\ V_1\ V_2 \cong_\rho W_i\ \rho\ V_1\ V_2$, or just $M\ \rho\ V_1\ V_2 \cong_\sigma V_i$ (**1**).

Let ρ and $V_1, V_2 : \rho$ be fixed. Consider R equal to $\{(W_1, V_1), (W_2, V_2)\}$ in $\mathcal{R}(\sigma\ \ , \rho)$ and η be $\alpha \mapsto (\sigma\ \ , \rho, R)$. We have $(W_1, V_1) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(W_1, V_1)$ and, similarly, $(W_2, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.

We have $(M, M) \in \mathcal{E}[\![\sigma]\!]$ by parametricity. Hence, $(M\ \sigma\ \ W_1\ W_2, M\ \rho\ V_1\ V_2)$ is in $\mathcal{V}[\![\alpha]\!]_\eta$, which means that $(M\ \sigma\ \ W_1\ W_2, M\ \rho\ V_1\ V_2)$ reduces to a pair of values in R, which implies:

$$\forall\rho, V_1, V_2, \quad \bigvee \begin{cases} \forall\rho, V_1, V_2, \ \ M\ \sigma\ \ W_1\ W_2 \cong_\sigma\ W_1 \ \wedge\ M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ \forall\rho, V_1, V_2, \ \ M\ \sigma\ \ W_1\ W_2 \cong_\sigma\ W_2 \ \wedge\ M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

Since, $M\ \sigma\ \ W_1\ W_2$ is independent of ρ, V_1, and V_2, this actually shows (1).

Exercise

Inhabitants of $\forall\alpha.\,\alpha \to \alpha$

Redo the proof that all inhabitants of of $\forall\alpha.\,\alpha \to \alpha$ are observationally equivalent to the identity, following the schema that we used for booleans.

Applications Inhabitants of $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

Fact Let *nat* be $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$. If $M : \mathit{nat}$, then $M \cong_{\mathit{nat}} N_n$ for some integer n, where $N_n \triangleq \Lambda\alpha.\,\lambda f{:}\alpha \to \alpha.\,\lambda x{:}\alpha.\,f^n\,x$.

Applications — Inhabitants of $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

Fact Let *nat* be $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$. If $M : \textit{nat}$, then $M \simeq_{\textit{nat}} N_n$ for some integer n, where $N_n \triangleq \Lambda\alpha.\,\lambda f{:}\alpha \to \alpha.\,\lambda x{:}\alpha.\,f^n\,x$.

Applications — Inhabitants of $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

Fact Let *nat* be $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$. If $M : \mathit{nat}$, then $M \cong_{\mathit{nat}} N_n$ for some integer n, where $N_n \triangleq \Lambda\alpha.\,\lambda f{:}\alpha \to \alpha.\,\lambda x{:}\alpha.\,f^n\,x$.

That is, the inhabitants of $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$ are the Church naturals.

Proof By *extensionality*, it suffices to show that there exists n such that for any closed type ρ and closed values $V_1 : \rho \to \rho$ and $V_2 : \rho$, we have $M\ \rho\ V_1\ V_2 \cong_\rho N_n\ \rho\ V_1\ V_2$, or, by closure by inverse reduction and replacing observational by logical equivalence, $M\ \rho\ V_1\ V_2 \sim_\rho V_1^n\ V_2$ $(\mathbf{1})$, since $N_n\ \rho\ V_1\ V_2$ reduces to $V_1^n\ V_2$. Let ρ and $V_1 : \rho \to \rho$ and $V_2 : \rho$ be fixed.

Let Z be $N_0\ \mathit{nat}$ and S be $N_1\ \mathit{nat}$. Let R in $\mathcal{R}(\mathit{nat}, \rho)$ be $\{(S^k\ Z, V_1^k\ V_2) \mid k \in \mathbb{N}\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.
We also have $(S, V_1) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$. (A key to the proof.)

Applications — Inhabitants of $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

Fact Let *nat* be $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$. If $M : \mathit{nat}$, then $M \cong_{\mathit{nat}} N_n$ for some integer n, where $N_n \triangleq \Lambda\alpha.\,\lambda f{:}\alpha \to \alpha.\,\lambda x{:}\alpha.\,f^n\,x$.

Proof By *extensionality*, it suffices to show that there exists n such that for any closed type ρ and closed values $V_1 : \rho \to \rho$ and $V_2 : \rho$, we have $M\;\rho\,V_1\;V_2 \cong_\rho N_n\;\rho\,V_1\;V_2$, or, by closure by inverse reduction and replacing observational by logical equivalence, $M\;\rho\,V_1\;V_2 \sim_\rho V_1^n\;V_2$ $(\mathbf{1})$, since $N_n\;\rho\,V_1\;V_2$ reduces to $V_1^n\;V_2$. Let ρ and $V_1 : \rho \to \rho$ and $V_2 : \rho$ be fixed.

Let Z be N_0 *nat* and S be N_1 *nat*. Let R in $\mathcal{R}(\mathit{nat}, \rho)$ be $\{(S^k\;Z, V_1^k\;V_2) \mid k \in \mathbb{N}\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.
We also have $(S, V_1) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$. (A key to the proof.)

Applications — Inhabitants of $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

Fact Let *nat* be $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$. If $M : \mathit{nat}$, then $M \cong_{\mathit{nat}} N_n$ for some integer n, where $N_n \triangleq \Lambda\alpha.\,\lambda f{:}\alpha \to \alpha.\,\lambda x{:}\alpha.\,f^n\,x$.

Proof By *extensionality*, it suffices to show that there exists n such that for any closed type ρ and closed values $V_1 : \rho \to \rho$ and $V_2 : \rho$, we have $M\;\rho\,V_1\;V_2 \cong_\rho N_n\;\rho\,V_1\;V_2$, or, by closure by inverse reduction and replacing observational by logical equivalence, $M\;\rho\,V_1\;V_2 \sim_\rho V_1^n\;V_2$ $(\mathbf{1})$, since $N_n\;\rho\,V_1\;V_2$ reduces to $V_1^n\;V_2$. Let ρ and $V_1 : \rho \to \rho$ and $V_2 : \rho$ be fixed.

Let Z be N_0 *nat* and S be N_1 *nat*. Let R in $\mathcal{R}(\mathit{nat},\rho)$ be $\{(S^k\;Z, V_1^k\;V_2) \mid k \in \mathbb{N}\}$ and η be $\alpha \mapsto (\mathit{nat},\rho,R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.
We also have $(S, V_1) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$. (A key to the proof.)

Applications — Inhabitants of $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

Fact Let *nat* be $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$. If $M : \mathit{nat}$, then $M \simeq_{\mathit{nat}} N_n$ for some integer n, where $N_n \triangleq \Lambda\alpha.\,\lambda f\!:\!\alpha \to \alpha.\,\lambda x\!:\!\alpha.\,f^n\,x$.

Proof By *extensionality*, it suffices to show that there exists n such that for any closed type ρ and closed values $V_1 : \rho \to \rho$ and $V_2 : \rho$, we have $M\ \rho\ V_1\ V_2 \simeq_\rho N_n\ \rho\ V_1\ V_2$, or, by closure by inverse reduction and replacing observational by logical equivalence, $M\ \rho\ V_1\ V_2 \sim_\rho V_1^n\ V_2$ $(\mathbf{1})$, since $N_n\ \rho\ V_1\ V_2$ reduces to $V_1^n\ V_2$. Let ρ and $V_1 : \rho \to \rho$ and $V_2 : \rho$ be fixed.

Let Z be N_0 *nat* and S be N_1 *nat*. Let R in $\mathcal{R}(\mathit{nat}, \rho)$ be $\{(S^k\ Z, V_1^k\ V_2) \mid k \in \mathbb{N}\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.
We also have $(S, V_1) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$. (A key to the proof.)

Applications — Inhabitants of $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

Fact Let *nat* be $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$. If $M : \mathit{nat}$, then $M \cong_{\mathit{nat}} N_n$ for some integer n, where $N_n \triangleq \Lambda\alpha.\,\lambda f{:}\alpha \to \alpha.\,\lambda x{:}\alpha.\,f^n\,x$.

Proof By *extensionality*, it suffices to show that there exists n such that for any closed type ρ and closed values $V_1 : \rho \to \rho$ and $V_2 : \rho$, we have $M\;\rho\,V_1\;V_2 \cong_\rho N_n\;\rho\,V_1\;V_2$, or, by closure by inverse reduction and replacing observational by logical equivalence, $M\;\rho\,V_1\;V_2 \sim_\rho V_1^n\;V_2$ $(\mathbf{1})$, since $N_n\;\rho\,V_1\;V_2$ reduces to $V_1^n\;V_2$. Let ρ and $V_1 : \rho \to \rho$ and $V_2 : \rho$ be fixed.

Let Z be $N_0\;\mathit{nat}$ and S be $N_1\;\mathit{nat}$. Let R in $\mathcal{R}(\mathit{nat}, \rho)$ be $\{(S^k\;Z, V_1^k\;V_2) \mid k \in \mathbb{N}\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.
We also have $(S, V_1) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$. (A key to the proof.)

Indeed, assume (W_1, W_2) in $\mathcal{V}[\![\alpha]\!]_\eta$. There exists k such that $W_1 = S^k\;Z$ and $W_2 = V_1^k\;V_2$. Thus, $(S\,W_1, V_1\;W_2)$ equal to $(S^{k+1}\;Z, V_1^{k+1}\;V_2)$ is in $\mathcal{E}[\![\alpha]\!]_\eta$.

Applications — Inhabitants of $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

Fact Let *nat* be $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$. If $M : \mathit{nat}$, then $M \simeq_{\mathit{nat}} N_n$ for some integer n, where $N_n \triangleq \Lambda\alpha.\,\lambda f{:}\alpha \to \alpha.\,\lambda x{:}\alpha.\,f^n\,x$.

Proof By *extensionality*, it suffices to show that there exists n such that for any closed type ρ and closed values $V_1 : \rho \to \rho$ and $V_2 : \rho$, we have $M\ \rho\,V_1\ V_2 \simeq_\rho N_n\ \rho\,V_1\ V_2$, or, by closure by inverse reduction and replacing observational by logical equivalence, $M\ \rho\,V_1\ V_2 \sim_\rho V_1^n\ V_2$ $(\mathbf{1})$, since $N_n\ \rho\,V_1\ V_2$ reduces to $V_1^n\ V_2$. Let ρ and $V_1 : \rho \to \rho$ and $V_2 : \rho$ be fixed.

Let Z be $N_0\ \mathit{nat}$ and S be $N_1\ \mathit{nat}$. Let R in $\mathcal{R}(\mathit{nat}, \rho)$ be $\{(S^k\ Z, V_1^k\ V_2) \mid k \in \mathbb{N}\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.
We also have $(S, V_1) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$. (A key to the proof.)

Indeed, assume (W_1, W_2) in $\mathcal{V}[\![\alpha]\!]_\eta$. There exists k such that $W_1 = S^k\ Z$ and $W_2 = V_1^k\ V_2$. Thus, $(S\,W_1, V_1\ W_2)$ equal to $(S^{k+1}\ Z, V_1^{k+1}\ V_2)$ is in $\mathcal{E}[\![\alpha]\!]_\eta$.

Applications

Inhabitants of $\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$

Fact Let *nat* be $\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$. If $M : \mathit{nat}$, then $M \cong_{\mathit{nat}} N_n$ for some integer n, where $N_n \triangleq \Lambda\alpha.\lambda f{:}\alpha \to \alpha.\lambda x{:}\alpha. f^n\, x$.

Proof By *extensionality*, it suffices to show that there exists n such that for any closed type ρ and closed values $V_1 : \rho \to \rho$ and $V_2 : \rho$, we have $M\ \rho\ V_1\ V_2 \cong_\rho N_n\ \rho\ V_1\ V_2$, or, by closure by inverse reduction and replacing observational by logical equivalence, $M\ \rho\ V_1\ V_2 \sim_\rho V_1^n\ V_2$ $(\mathbf{1})$, since $N_n\ \rho\ V_1\ V_2$ reduces to $V_1^n\ V_2$. Let ρ and $V_1 : \rho \to \rho$ and $V_2 : \rho$ be fixed.

Let Z be $N_0\ \mathit{nat}$ and S be $N_1\ \mathit{nat}$. Let R in $\mathcal{R}(\mathit{nat}, \rho)$ be $\{(S^k\ Z, V_1^k\ V_2) \mid k \in \mathbb{N}\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$.
We also have $(S, V_1) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$. (A key to the proof.)

By parametricity, we have $M \sim_{\mathit{nat}} M$. Hence, $(M\ \mathit{nat}\ S\ Z, M\ \rho\ V_1\ V_2) \in \mathcal{E}[\![\alpha]\!]_\eta$. Thus, there exists n such that $M\ \mathit{nat}\ S\ Z \cong_{\mathit{nat}} S^n\ Z$ and $M\ \rho\ V_1\ V_2 \cong_\rho V_1^n\ V_2$.

Since, $M\ \mathit{nat}\ S\ Z$ is independent of n, we may conclude (1), provided the $S^n\ Z$ are all in different observational equivalence classes (easy to check).

Applications *Inhabitants of* $\forall \alpha. \alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

- Left as an exercise...

Applications $\quad \forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

Fact Let τ be closed and *list* be $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$. Let C be $\lambda H{:}\tau.\,\lambda T{:}\mathit{list}\,.\,\Lambda\alpha.\,\lambda n{:}\alpha.\,\lambda c{:}\tau \to \alpha \to \alpha.\,c\,H\,(T\;\alpha\;n\;c)$ and N be $\Lambda\alpha.\,\lambda n{:}\alpha.\,\lambda c{:}\tau \to \alpha \to \alpha.\,n$. If $M : \mathit{list}$, then $M \cong_{\mathit{list}} N_n$ for some N_n in $\mathcal{L}_n$ where $\mathcal{L}_k$ is defined inductively by

$$\mathcal{L}_0 \triangleq \{N\} \text{ and } \mathcal{L}_{k+1} \triangleq \{C\,W_k\;N_k \mid W_k \in \mathrm{Val}(\tau) \wedge N_k \in \mathcal{L}_k\}$$

Proof By *extensionality*, it suffices to show that there exists n and $N_n \in \mathcal{L}_n$ such that for any closed type ρ and closed values $V_1 : \tau \to \rho \to \rho$ and $V_2 : \rho$, we have $M\;\rho\;V_1\;V_2 \sim_\rho N_n\;\rho\;V_1\;V_2$, or, by closure by inverse reduction and replacing observational by logical equivalence, $C\;W_n\;(\ldots(C\;W_1\;N)\ldots)$ (**1**), since $N_n\;\rho\;V_1\;V_2$ reduces to $C\;W_n\;(\ldots(C\;W_1\;N)\ldots)$ where all W_k are in $\mathrm{Val}(\tau)$.

Let ρ and $V_1 : \alpha \to \rho \to \rho$ and $V_2 : \rho$ be fixed.

Let R in $\mathcal{R}(\mathit{list}, \rho)$ be defined inductively as $\bigcup R_n$ where R_{k+1} is $\{\Downarrow (C\,G\;T, V_2\;H\;U) \mid (G,H) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T,U) \in R_k\}$ and R_0 is $\{(N, V_1)\}$.

We have $(N, V_1) \in R_0 \subseteq \mathcal{V}[\![\alpha]\!]_\eta$.

We also have $(C,\;V_2) \in \mathcal{V}[\![\tau \to \alpha \to \alpha]\!]_\eta$. (A key to the proof)

Applications $\quad \forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

Fact Let τ be closed and *list* be $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$. Let C be $\lambda H{:}\tau.\,\lambda T{:}\mathit{list}\,.\,\Lambda\alpha.\,\lambda n{:}\alpha.\,\lambda c{:}\tau \to \alpha \to \alpha.\,c\,H\;(T\;\alpha\;n\;c)$ and N be $\Lambda\alpha.\,\lambda n{:}\alpha.\,\lambda c{:}\tau \to \alpha \to \alpha.\,n$. If $M : \mathit{list}$, then $M \cong_{\mathit{list}} N_n$ for some N_n in $\mathcal{L}_n$ where $\mathcal{L}_k$ is defined inductively by

$$\mathcal{L}_0 \triangleq \{N\} \text{ and } \mathcal{L}_{k+1} \triangleq \{C\,W_k\;N_k \mid W_k \in \mathrm{Val}(\tau) \wedge N_k \in \mathcal{L}_k\}$$

Proof By *extensionality*, it suffices to show that there exists n and $N_n \in \mathcal{L}_n$ such that for any closed type ρ and closed values $V_1 : \tau \to \rho \to \rho$ and $V_2 : \rho$, we have $M\;\rho\;V_1\;V_2 \sim_\rho N_n\;\rho\;V_1\;V_2$, or, by closure by inverse reduction and replacing observational by logical equivalence, $C\;W_n\;(\ldots(C\;W_1\;N)\ldots)$ $(\mathbf{1})$, since $N_n\;\rho\;V_1\;V_2$ reduces to $C\;W_n\;(\ldots(C\;W_1\;N)\ldots)$ where all W_k are in $\mathrm{Val}(\tau)$.

Let ρ and $V_1 : \alpha \to \rho \to \rho$ and $V_2 : \rho$ be fixed.

Let R in $\mathcal{R}(\mathit{list}, \rho)$ be defined inductively as $\bigcup R_n$ where R_{k+1} is $\{\Downarrow (C\,G\;T, V_2\;H\;U) \mid (G, H) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T, U) \in R_k\}$ and R_0 is $\{(N, V_1)\}$.

We have $(N, V_1) \in R_0 \subseteq \mathcal{V}[\![\alpha]\!]_\eta$.

We also have $(C,\;V_2) \in \mathcal{V}[\![\tau \to \alpha \to \alpha]\!]_\eta$. $\qquad$ (A key to the proof)

Applications $\quad \forall\alpha.\, \alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

Fact Let τ be closed and *list* be $\forall\alpha.\, \alpha \to (\tau \to \alpha \to \alpha) \to \alpha$. Let C be $\lambda H{:}\tau.\, \lambda T{:}\mathit{list}\,.\, \Lambda\alpha.\, \lambda n{:}\alpha.\, \lambda c{:}\tau \to \alpha \to \alpha.\, c\; H\; (T\; \alpha\; n\; c)$ and N be $\Lambda\alpha.\, \lambda n{:}\alpha.\, \lambda c{:}\tau \to \alpha \to \alpha.\, n$. If $M : \mathit{list}$, then $M \cong_{\mathit{list}} N_n$ for some N_n in $\mathcal{L}_n$ where $\mathcal{L}_k$ is defined inductively by

$$\mathcal{L}_0 \triangleq \{N\} \text{ and } \mathcal{L}_{k+1} \triangleq \{C\, W_k\; N_k \mid W_k \in \mathrm{Val}(\tau) \wedge N_k \in \mathcal{L}_k\}$$

Proof By *extensionality*, it suffices to show that there exists n and $N_n \in \mathcal{L}_n$ such that for any closed type ρ and closed values $V_1 : \tau \to \rho \to \rho$ and $V_2 : \rho$, we have $M\; \rho\; V_1\; V_2 \sim_\rho N_n\; \rho\; V_1\; V_2$, or, by closure by inverse reduction and replacing observational by logical equivalence, $C\;\; W_n\; (\ldots (C\;\; W_1\;\; N) \ldots)$ (**1**), since $N_n\; \rho\; V_1\; V_2$ reduces to $C\;\; W_n\; (\ldots (C\;\; W_1\;\; N) \ldots)$ where all W_k are in $\mathrm{Val}(\tau)$.

Let ρ and $V_1 : \alpha \to \rho \to \rho$ and $V_2 : \rho$ be fixed.

Let R in $\mathcal{R}(\mathit{list}, \rho)$ be defined inductively as $\bigcup R_n$ where R_{k+1} is $\{\Downarrow (C\, G\; T, V_2\; H\; U) \mid (G, H) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T, U) \in R_k\}$ and R_0 is $\{(N, V_1)\}$.

We have $(N, V_1) \in R_0 \subseteq \mathcal{V}[\![\alpha]\!]_\eta$.

We also have $(C,\; V_2) \in \mathcal{V}[\![\tau \to \alpha \to \alpha]\!]_\eta$. (A key to the proof)

Contents

Encodable features — Natural numbers

We have shown that all expressions of type *nat* behave as natural numbers. Hence, natural numbers are definable.

Still, we could also provide a type *nat* of natural numbers as primitive.

Then, we may extend

- behavioral equivalence: if $M_1 : \mathit{nat}$ and $M_2 : \mathit{nat}$, we have $M_1 \simeq_{\mathit{nat}} M_2$ iff there exists $n : \mathit{nat}$ such that $M_1 \Downarrow n$ and $M_2 \Downarrow n$.
- logical equivalence: $\mathcal{V}[\![\mathit{nat}]\!] \triangleq \{(n, n) \mid n \in \mathbb{N}\}$

All properties are preserved.

Encodable features

Products

Given closed types τ_1 and τ_2, we defined

$$\begin{aligned} \tau_1 \times \tau_2 &\triangleq \forall \alpha. (\tau_1 \to \tau_2 \to \alpha) \to \alpha \\ (M_1, M_2) &\triangleq \Lambda \alpha. \lambda x{:}\tau_1 \to \tau_2 \to \alpha. x\ M_1\ M_2 \\ M.i &\triangleq M\ (\lambda x_1{:}\tau_1. \lambda x_2{:}\tau_2. x_i) \end{aligned}$$

Facts

If $M : \tau_1 \times \tau_2$, then $M \cong_{\tau_1 \times \tau_2} (M_1, M_2)$ for some $M_1 : \tau_1$ and $M_2 : \tau_2$.

If $M : \tau_1 \times \tau_2$ and $M.1 \cong_{\tau_1} M_1$ and $M.2 \cong_{\tau_2} M_2$, then $M \cong_{\tau_1 \times \tau_2} (M_1, M_2)$

Primitive pairs

We may instead extend the language with *primitive* pairs. Then,

$$\mathcal{V}[\![\tau \times \sigma]\!]_\eta \triangleq \left\{\big((V_1, W_1), (V_2, W_2)\big) \;\middle|\; (V_1, V_2) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (W_1, W_2) \in \mathcal{V}[\![\sigma]\!]_\eta\right\}$$

Sums

We define:

$$\begin{array}{rcl} \mathcal{V}[\![\tau + \sigma]\!]_\eta & = & \{(\mathit{inj}_1\ V_1, \mathit{inj}_1\ V_2) \mid (V_1, V_2) \in \mathcal{V}[\![\tau]\!]_\eta\} \ \cup \\ & & \{(\mathit{inj}_2\ W_1, \mathit{inj}_2\ W_2) \mid (W_1, W_2) \in \mathcal{V}[\![\sigma]\!]_\eta\} \end{array}$$

Notice that sums, as all datatypes, can also be encoded in System F.

Primitive Lists

We *recursively*[1] define $\mathcal{V}[\![\mathit{list}\,\tau]\!]_\eta$ as $\bigcup_k \mathcal{W}_\eta^k$ where $\mathcal{W}_\eta^0$ is $\{(\mathit{Nil}, \mathit{Nil})\}$ and $\mathcal{W}_\eta^{k+1}$ is
$\{(\mathit{Cons}\ H_1\ T_1, \mathit{Cons}\ H_2\ T_2) \mid (H_1, H_2) \in \mathcal{V}[\![\alpha]\!]_\eta \ \wedge\ (T_1, T_2) \in \mathcal{W}_\eta^k\}$.
Assume that $(\alpha \mapsto \rho_1, \rho_2, R) \in \eta$ where R in $\mathcal{R}(\rho_1, \rho_2)$ is the graph $\langle g \rangle$ of a function g, *i.e.* equal to $\{(V_1, V_2) \mid g\, V_1 \Downarrow V_2\}$. Then, we have:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{V}[\![\mathit{list}\,\alpha]\!]_\eta(W_1, W_2)} \\
\iff & \exists k, \bigvee \left\{ \begin{array}{l} W_1 = \mathit{Nil} \wedge W_2 = \mathit{Nil} \\ W_1 = \mathit{Cons}\ H_1\ T_1 \wedge W_2 = \mathit{Cons}\ H_2\ T_2 \wedge g\ H_1 \Downarrow H_2 \\ \qquad \wedge\ (T_1, T_2) \in \mathcal{W}_\eta^k \end{array} \right. \\
\iff & \mathit{map}\ \rho_1\ \rho_2\ g\ \ W_1 \Downarrow W_2
\end{array}
$$

[1] This definition is well-founded.

Applications $\mathit{sort} : \forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha$

Fact: Assume $\mathit{sort} : \forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha \to \mathit{list}\,\alpha$ $(\mathbf{1})$. Then

$$(\forall x, y,\ \ \mathit{cmp}_2\ (f\ x)\ (f\ y) = \mathit{cmp}_1\ \ x\ y) \implies \forall \ell,\ \ \mathit{sort}\ \ \mathit{cmp}_2\ (\mathit{map}\ f\ \ell) = \mathit{map}\ f\ (\mathit{sort}\ \mathit{cmp}_1\ \ell)$$

Applications $\quad$ *sort* $: \forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\;\alpha$

Proof: Assume $\forall x, y,\;\; \mathit{cp}\;(f\;x)\;(f\;y) \cong \mathit{cp}\;\;x\;y)$ (**H**).

We have $\mathit{sort} \sim_\sigma \mathit{sort}$ where σ is $\forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\;\alpha \to \mathit{list}\;\alpha$.

Thus, for all ρ_1, ρ_2, and relations R in $\mathcal{R}(\rho_1, \rho_2)$,

$$\forall(\mathit{cp}_1, \mathit{cp}_2) \in \mathcal{V}[\![\alpha \to \alpha \to \mathsf{B}]\!]_\eta, \qquad (\mathbf{1})$$

$$\forall(V_1, V_2) \in \mathcal{V}[\![\mathit{list}\;\alpha]\!]_\eta,\;\; (\mathit{sort}\;\rho_1\;\mathit{cp}_1\;\;V_1, \mathit{sort}\;\rho_2\;\mathit{cp}_2\;\;V_2) \in \mathcal{E}[\![\mathit{list}\;\alpha]\!]_\eta) \qquad (\mathbf{2})$$

where η is $\alpha \mapsto (\rho_1, \rho_2, R)$. We may choose R to be $\langle f \rangle$ for some f.

We have (1). Indeed, for all (V_1, V_2) and (W_1, W_2) in $\langle f \rangle$, we have $f\;V_1 \Downarrow V_1$ and $f\;W_1 \Downarrow W_1$, hence $\mathit{cp}_2\;(f\;V_1)(f\;W_1) \Downarrow \mathit{cp}_1\;V_2 W_2$. Thus $\mathit{cp}_2\;(f\;V_1)(f\;W_1) \cong \mathit{cp}_1\;V_2 W_2$. With (H), this implies $\mathit{cp}_2\;V_1 W_1 \cong \mathit{cp}_1\;V_2 W_2$, *i.e.* $\mathit{cp}_2\;V_1 W_1 \sim \mathit{cp}_1\;V_2 W_2$ since we are at type B, as expected. Hence (2) holds.

Since

$$\mathcal{V}[\![\mathit{list}\;\alpha]\!]_\eta \;\triangleq\; \langle \mathit{map}\;\rho_1\;\rho_2\;f \rangle \;\subseteq\; \mathcal{V}[\![\rho_1]\!] \times \mathcal{V}[\![\rho_2]\!]$$

(2) reads

$$\begin{array}{l} \forall V\;\;: \mathit{list}\;\rho_1, V_2 :: \mathit{list}\;\rho_2, \\ \qquad \mathit{map}\;\rho_1\;\rho_2\;f\;V \Downarrow V_2 \implies \exists W_1, W_2, \left\{ \begin{array}{l} \mathit{map}\;\rho_1\;\rho_2\;f\;W_1 \\ \mathit{sort}\;\rho_1\;\mathit{cp}_1\;\;V \Downarrow W_1 \qquad \cong \\ \mathit{sort}\;\rho_2\;\mathit{cp}_2\;V_2 \end{array} \right. \end{array}$$

Applications *whoami* $: \forall \alpha.\, \textit{list}\ \alpha \to \textit{list}\ \alpha$

Left as an exercise...

Existential types

We define:

$$\mathcal{V}[\![\exists\alpha.\,\tau]\!]_\eta \triangleq \Big\{(\textit{pack}\ V_1, \rho_1\ \textit{as}\ \exists\alpha.\,\tau, \textit{pack}\ V_2, \rho_2\ \textit{as}\ \exists\alpha.\,\tau) \mid \exists \rho_1, \rho_2, R \in \mathcal{R}(\rho_1, \rho_2),\ \ (V_1, V_2) \in \mathcal{E}[\![\tau]\!]_{\eta, \alpha \mapsto (\rho_1, \rho_2, R)}\Big\}$$

Compare with

$$\mathcal{V}[\![\forall\alpha.\,\tau]\!]_\eta = \Big\{(\Lambda\alpha.\,M_1, \Lambda\alpha.\,M_2) \mid \forall \rho_1, \rho_2, R \in \mathcal{R}(\rho_1, \rho_2),\ ((\Lambda\alpha.\,M_1)\ \rho_1, (\Lambda\alpha.\,M_2)\ \rho_2) \in \mathcal{E}[\![\tau]\!]_{\eta, \alpha \mapsto (\rho_1, \rho_2, R)}\Big\}$$

Existential types Example

Consider $V_1 \triangleq (\textit{not}, \mathrm{tt})$, and $V_2 \triangleq (\textit{succ}, 0)$ and $\sigma \triangleq (\alpha \to \alpha) \times \alpha$.
Let $R \in \mathcal{R}(\textit{bool}, \textit{nat})$ be $\{(\mathrm{tt}, 2n), (\mathrm{ff}, 2n+1) \mid n \in \mathbb{N}\}$ and η be $\alpha \mapsto (\textit{bool}, \textit{nat}, R)$.

We have $(V_1, V_2) \in \mathcal{V}[\![\sigma]\!]_\eta$.

Hence, $(\textit{pack}\, V_1, \textit{bool as}\ \exists\alpha.\,\sigma,\ \textit{pack}\, V_2, \textit{nat as}\ \exists\alpha.\,\sigma) \in \mathcal{V}[\![\exists\alpha.\,\sigma]\!]$.

Proof of $\big((\textit{not}, \mathrm{tt}), (\textit{succ}, 0)\big) \in \mathcal{V}[\![(\alpha \to \alpha) \times \alpha]\!]_\eta$ **(1)**

We have $(\mathrm{tt}, 0) \in \mathcal{V}[\![\alpha]\!]_\eta$, since $(\mathrm{tt}, 0) \in R$.
We also have $(\textit{not}, \textit{succ}) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$, which proves (1).

Indeed, assume $(W_1, W_2) \in \mathcal{V}[\![\alpha]\!]_\eta$. Then (W_1, W_2) is either of the form

- $(\mathrm{tt}, 2n)$ and $(\textit{not}\, W_1, \textit{succ}\, W_2)$ reduces to $(\mathrm{ff}, 2n+1)$, or
- $(\mathrm{ff}, 2n+1)$ and $(\textit{not}\, W_1, \textit{succ}\, W_2)$ reduces to $(\mathrm{tt}, 2n+2)$.

In both cases, $(\textit{not}\, W_1, \textit{succ}\, W_2)$ reduces to a pair in R.
Hence, $(\textit{not}\, W_1, \textit{succ}\, W_2) \in \mathcal{E}[\![\alpha]\!]_\eta$.

Representation independence

A client of an existential type $\exists\alpha.\,\tau$ should not see the difference between two implementations N_1 and N_2 of $\exists\alpha.\,\tau$ with witness types ρ_1 and ρ_2.

A client M has type $\forall\alpha.\,\tau \to \sigma$ with $\alpha \notin \mathrm{fv}(\sigma)$; it must use the argument parametrically, and the result is independent of the witness type.

Assume that ρ_1 and ρ_2 are two closed representation types and R is in $\mathcal{R}(\rho_1, \rho_2)$. Let η be $\alpha \mapsto (\rho_1, \rho_2, R)$.

Suppose that $N_1 : \tau[\alpha \mapsto \rho_1]$ and $N_2 : \tau[\alpha \mapsto \rho_2]$ are two equivalent implementations of the operations, *i.e.* such that $(N_1, N_2) \in \mathcal{E}[\![\tau]\!]_\eta$.

A client M satisfies $(M, M) \in \mathcal{E}[\![\forall\alpha.\,\tau \to \sigma]\!]_\eta$. Thus $(M\ \rho_1\ N_1, M\ \rho_2\ N_2)$ is in $\mathcal{E}[\![\sigma]\!]$ (as α is not free in σ).

That is, $M\ \rho_1\ N_1 \cong_\sigma M\ \rho_2\ N_2$: the behavior with the implementation N_1 with representation type ρ_1 is indistinguishable from the behavior with the implementation N_2 with representation type ρ_2.

How do we deal with recursive types?

Assume that we allow equi-recursive types.

$$\tau ::= \ldots \mid \mu\alpha.\tau$$

A naive definition would be

$$\mathcal{V}[\![\mu\alpha.\tau]\!]_\eta = \mathcal{V}[\![[\alpha \mapsto \mu\alpha.\tau]\tau]\!]_\eta$$

But this is ill-founded.

The solution is to use indexed-logical relations.

We use a sequence of decreasing relations indexed by integers (fuel), which is consumed during unfolding of recursive types.

Step-indexed logical relations (a taste)

We define a sequence $\mathcal{V}_k[\![\tau]\!]_\eta$ indexed by natural numbers $n \in \mathbb{N}$ that relates values of type τ up to n reduction steps. Omitting typing clauses:

$$
\begin{array}{rcl}
\mathcal{V}_k[\![\mathsf{B}]\!]_\eta & = & \{(\mathsf{tt}, \mathsf{tt}), (\mathsf{ff}, \mathsf{ff})\} \\
\mathcal{V}_k[\![\tau \to \sigma]\!]_\eta & = & \{(V_1, V_2) \mid \forall j < k, \forall (W_1, W_2) \in \mathcal{V}_j[\![\tau]\!]_\eta, \\
& & \quad (V_1\ W_1, V_2\ W_2) \in \mathcal{E}_j[\![\sigma]\!]_\eta\} \\
\mathcal{V}_k[\![\alpha]\!]_\eta & = & \eta_R(\alpha).k \\
\mathcal{V}_k[\![\forall\alpha.\,\tau]\!]_\eta & = & \big\{(V_1, V_2) \mid \forall \rho_1, \rho_2, R \in \mathcal{R}^k(\rho_1, \rho_2), \forall j < k, \\
& & \quad (V_1\ \rho_1, V_2\ \rho_2) \in \mathcal{V}_j[\![\tau]\!]_{\eta, \alpha \mapsto (\rho_1, \rho_2, R)}\big\} \\
\mathcal{V}_k[\![\mu\alpha.\tau]\!]_\eta & = & \mathcal{V}_{k-1}[\![[\alpha \mapsto \mu\alpha.\tau]\tau]\!]_\eta \\
\mathcal{E}_k[\![\tau]\!]_\eta & = & \{(M_1, M_2) \mid \forall j < k, M_1 \Downarrow_j V_1 \\
& & \quad \Longrightarrow \exists V_2, M_2 \Downarrow V_2 \wedge (V_1, V_2) \in \mathcal{V}_{k-j}[\![\tau]\!]_\eta\}
\end{array}
$$

By $\Downarrow_j$ means *reduces in j-steps*.
$\mathcal{R}^j(\rho_1, \rho_2)$ is composed of sequences of decreasing relations between closed values of closed types ρ_1 and ρ_2 of length (at least) j.

Step-indexed logical relations (a taste)

The relation is asymmetric.

If $\Delta;\Gamma \vdash M_1, M_2 : \tau$ we define $\Delta;\Gamma \vdash M_1 \precsim M_2 : \tau$ as

$$\forall \eta \in \mathcal{R}^k_\Delta(\delta_1,\delta_2), \forall(\gamma_1,\gamma_2) \in \mathcal{G}_k[\![\Gamma]\!],\ \ (\gamma_1(\delta_1(M_1)),\gamma_2(\delta_2(M_2))) \in \mathcal{E}_k[\![\tau]\!]_\eta$$

and

$$\Delta;\Gamma \vdash M_1 \sim M_2 : \tau \ \ \triangleq\ \ \bigwedge\begin{cases}\Delta;\Gamma \vdash M_1 \precsim M_2 : \tau \\ \Delta;\Gamma \vdash M_2 \precsim M_1 : \tau\end{cases}$$

Notations and proofs get a bit involved...

Notations may be simplified by introducing a *later* guard $\triangleright$ to capture incrementation of the index and avoid the explicit manipulation of integers (but the meaning remains the same).

Logical relations for F^ω ?

Logical relations can be generalized to work for F^ω, indeed.

There is a slight complication though in the interpretation of type functions.

This is out of this course scope, but one may, for instance, read [Atkey, 2012].

Side effects, References, Value restriction

Contents

Referential transparency

What is it?

An expression is *referentially transparent* or *pure* if it can be replaced with its corresponding value without changing the program behavior. Applying a pure funtion to the same arguments returns the same result.

Why is it useful?

Allows to reason about programs as a rewrite system, which may help

- prove the correction,
- perform code optimization.
- typically, it allows for: memoization, common expression elimination, lazy evaluation, ...
- with code parallelization, optimistic evaluation, transactions, ...

Referential transparency *counter examples*

Examples of impure constructs

- Exceptions, References, reading/printing functions.
- Interaction with the file system.
- Date and random primitives, etc.

Termination?

According to the definition, the status of termination is unclear. (As they never return, they cannot actually be replaced by the result of their evaluation—except in Haskell that uses an explicit bottom value $\perp$.) Non-termination is usually considered impure: it breaks equational reasoning and most program transformations, as other impure constructs.

In practice, high-complexity is not so different from non-termimation...

Effects

Any source of impurity is usually called an *effect*.

Referential transparency *Summary*

Effects are unavoidable

Any programming language must have some impure aspects to communicate with the operating system.

Side effects may sometimes be encapsulated, *e.g.* a module with side effetcs may sometimes have a pure interface.

Mitigation of effects

So the questions are more whether:

- a large core of the language is pure/effect free (*e.g.* Haskell, Coq, Core System F) or effectful (most other languages); and/or
- side effects can be tracked, *e.g.* by the type system. (Haskell, Koka, Rust, Mezzo, or algebraic effects)

The semantics of effects

Programs with effects cannot be described as a pure rewrite system.

- The semantics must be changed.
- Some of the properties will be lost

We shall see:

- Exceptions, which require a small change to the semantics
- References, which:
  - require a major change to the semantics
  - do not fit well with polymorphism—which needs to be restricted in the presence of effects.
- Values, or a larger class of *non-expansive expressions*, whose evaluation is effect free play a key role in the presence of effects.

In the presence of effects, deterministic, call-by-value semantics is always a huge source of simplification when not a requirement.

Contents

Exceptions

Semantics

Exceptions are a mechanism for changing the normal order of evaluation usually, but not necessarily, in case something abnormal occurred.

When an exception is raised, the evaluation does not continue as usual: Shortcutting normal evaluation rules, the exception is propagated up into the evaluation context until some handler is found at which the evaluation resumes with the exceptional value received; if no handler is found, the exception had reached the toplevel and the result of the evaluation is the exception instead of a value.

We extend the language with a constructor form to raise an exception and a destructor form to catch an exception; we also extend the evaluation contexts:

$$\begin{array}{rcl} M & ::= & \ldots \mid \textit{raise}\ M \mid \textit{try}\ M\ \textit{with}\ M \\ E & ::= & \ldots \mid \textit{raise}\ [\,] \mid \textit{try}\ [\,]\ \textit{with}\ M \end{array}$$

Exceptions — Semantics

We do not treat *raise* V as a value, since it stops the normal order of evaluation. Instead, reduction rules propagate and handle exceptions:

RAISE
$$F[\textit{raise}\,V] \longrightarrow \textit{raise}\,V$$

HANDLE-VAL
$$\textit{try}\,V\ \textit{with}\ M \longrightarrow V$$

HANDLE-RAISE
$$\textit{try raise}\,V\ \textit{with}\ M \longrightarrow M\ V$$

Rule RAISE uses an evaluation context F which stands for *any E other than try $[]$ with M*, so that it propagates an exception up the evaluation contexts, but not through a handler.

The case of the handler is treated by two specific rules:

- Rule HANDLE-RAISE passes an exceptional value to its handler;
- Rule HANDLE-VAL removes the handler around a value.

Exceptions — Example

For example, assuming that K is $\lambda x.\, \lambda y.\, y$ and $M \longrightarrow V$, we have the following reduction:

$$
\begin{array}{llr}
 & \textit{try}\ K\ (\textit{raise}\ M)\ \textit{with}\ \lambda x.\, x & \text{by } \textsc{Context} \\
\longrightarrow & \textit{try}\ K\ (\textit{raise}\ V)\ \textit{with}\ \lambda x.\, x & \text{by } \textsc{Raise} \\
\longrightarrow & \textit{try}\ \textit{raise}\ V\ \textit{with}\ \lambda x.\, x & \text{by } \textsc{Handle-Raise} \\
\longrightarrow & (\lambda x.\, x)\ V & \text{by } \beta_v \\
\longrightarrow & V &
\end{array}
$$

In particular, we do not have the following step,

$$
\begin{array}{llr}
 & \textit{try}\ K\ (\textit{raise}\ V)\ \textit{with}\ \lambda x.\, x & \text{by } \beta_v \\
\not\longrightarrow & \textit{try}\ \lambda y.\, y\ \textit{with}\ \lambda x.\, x \longrightarrow \lambda y.\, y &
\end{array}
$$

since *raise* V is *not* a value, so the first β-reduction step is not allowed.

Exceptions — Typing rules

We assume given a *fixed type* τ_{exn} for exceptional values.

$$\textsc{Raise}\quad \frac{\Gamma \vdash M : \tau_{exn}}{\Gamma \vdash \textit{raise}\; M : \tau} \qquad\qquad \textsc{Try}\quad \frac{\Gamma \vdash M_1 : \tau \qquad \Gamma \vdash M_2 : \tau_{exn} \to \tau}{\Gamma \vdash \textit{try}\; M_1 \;\textit{with}\; M_2 : \tau}$$

There are some subtleties:

- Raise turns an expression of type τ_{exn} into an exception.
- Consistently, the handler has type $\tau_{exn} \to \tau$, since it receives the exception value of type *exn* as argument;
- An exceptional value of type *exn* may be raised in M_1 and used in M_2 without any visible flow at the type level.
  Hence, *raise* · and *try* · *with* · must agree on the type *exn*.
- Both premises of Rule Try must return values of the same type τ.
- *raise* M can have any type, as the current computation is aborted.

Exceptions — The type of exception

What should we choose for τ_{exn}? Well, any type:

- Choosing *unit*, exceptions will not carry any information.
- Choosing *int*, exceptions can report some error code.
- Choosing *string*, exceptions can report error messages.
- Using a sum type or better a variant type (tagged sum), with one case to describe each exceptional situation.

  This is the approach followed by ML, which declares a new extensible type *exn* for exceptions: this is a sum type, except that all cases are not declared in advance, but only as needed.
  (Extensible datatypes are available in OCaml since version 4.02.)

In all cases, the type of exception must be fixed in the whole program.

This is because *raise* · and *try* · *with* · must agree beforehand on the type of exceptions as this type is not passed around by the typing rules.

Encoding of multiple exceptions

Introduce a data type:

$$\text{type } \mathit{exn} = \Sigma(E_i : \tau_i \to \mathit{exn})^{i \in I}$$

Use syntactic sugar:

$$\mathit{raise}\ E_i\ v \ \triangleq\ \mathit{raise}\ (E_i\ v)$$

$$\begin{aligned} &\mathit{try}\ M\ \mathit{with}\ (E_j\ x \Rightarrow M_k)^{j \in J} \\ &\qquad \triangleq\ \mathit{try}\ M\ \mathit{with} \\ &\qquad\qquad (\lambda z.\, \mathit{match}\ z\ \mathit{with}\ (E_j\ x \Rightarrow M_k)^{j \in J} \mid z \Rightarrow \mathit{raise}\ z) \end{aligned}$$

Exceptions — Type soundness

How do we state type soundness, since exceptions may be uncaught?

By saying that this is the only "exception" to progress:

Theorem (Progress)

A well-typed, irreducible term is either a value or an uncaught exception.
if $\varnothing \vdash M : \tau$ *and* $M \not\longrightarrow$ *, then* M *is either* V *or raise* V *for some value* V*.*

Exceptions — Structured exceptions

What is the type *exn* for exceptions? Well, it could be any type:

- If we take the unit type, we only know that an except has been raised but cannot pass any other information.
- Hence, we could take the type of integers (*e.g.* passing error codes, much as commands do in Unix)
- Use some richer data type with one constructor per kind of error.
- Use a variant type (tagged sum)

The handler may analyze the argument of the exception.

Exceptions On uncaught exceptions

An uncaught exception is often a programming error. It may be surprising that they are not detected by the type system.

Exceptions may be detected using more expressive type systems. Unfortunately, the existing solutions are often complicated for some limited benefit, and are still not often used in practice.

The complication comes from the treatment of functions, which have some *latent effect* of possibly raising or catching an exception when applied. To be precise, the analysis must therefore enrich types of functions with latent effects, which is quite invasive and obfuscating.

Uncaught exceptions must be declared in the language Java. (Java also has untraced exceptions.)

See Leroy and Pessaux [2000] for a solution in ML.

Exceptions — Small semantic variation

Once raised, exceptions are propagated step-by-step by Rule RAISE until they reach a handler or the toplevel.

We can also describe their semantics by replacing propagation of exceptions by deep handling of exceptions inside terms.

Replace the three previous reduction rules by:

HANDLE-VAL'

$$\textit{try}\, V \;\textit{with}\, M \longrightarrow V$$

HANDLE-RAISE'

$$\textit{try}\, \bar{F}[\textit{raise}\, V]\; \textit{with}\, M \longrightarrow M\ V$$

where $\bar{F}$ is a sequence of F contexts, *i.e.* handler-free evaluation context of arbitrary depth.

This semantics is perhaps more intuitive, closer to what a compiler does, but the two presentations are equivalent.

In this case, uncaught exceptions are of the form $\bar{F}[\textit{raise}\, V]$.

Exceptions — Interesting syntactic variation

Benton and Kennedy [2001] have argued for merging let and try constructs into a unique form *let* $x = M_1$ *with* M_2 *in* M_3.

The expression M_1 is evaluated first and

- if it returns a value it is substituted for x in M_3, as if we had evaluated *let* $x = M_1$ *in* M_3;
- otherwise, *i.e.*, if it raises an exception *raise* V, then the exception is handled by M_2, as if we had evaluated *try* M_1 *with* M_2.

This combined form captures a common programming pattern:

```
let rec read_config_in_path filename (dir :: dirs) →
  let fd = open_in (Filename.concat dir filename)
  with Sys_error _ → read_config filename dirs in
  read_config_from_fd fd
```

Workarounds are inelegant and inefficient. This form is also better suited for program transformations (see Benton and Kennedy [2001]).

Exceptions — Interesting syntactic variation

Encoding the new form *let* x = M_1 *with* M_2 *in* M_3 with "let" and "try" is not easy:

In particular, it is not equivalent to: *try let* $x = M_1$ *in* M_3 *with* M_2.

The continuation M_3 could raise an exception that would then be handled by M_2, which is not intended.

There are several encodings:

- Use a sum type to know whether M_1 raised an exception:
  case (*try Val* M_1 *with* $\lambda y.$ *Exc* y) *of* (*Val* : $\lambda x. M_3$ [] *Exc* : M_2)
- Freeze the continuation M_3 while handling the exception:
  (*try let* $x = M_1$ *in* $\lambda(). M_3$ *with* $\lambda y. \lambda(). M_2\ y$) ()

Unfortunately, they are both hardly readable—and inefficient.

Exceptions — Interesting syntactic variation

A similar construct has been added in OCaml version 4.02, allowing exceptions combined with pattern matching.

The previous example can now be written in OCaml as:

$$
\begin{array}{l}
\textbf{let rec } \textit{read\_config\_in\_path filename path} = \\
\quad \textbf{match } \textit{path } \textbf{with } \ [] \to [] \mid \textit{dir} :: \textit{dirs} \to \\
\qquad \textbf{match } \textit{open\_in } (\textit{Filename.concat dir filename}) \textbf{ with} \\
\qquad \mid \textit{fd} \to \textit{read\_config\_from\_fd fd} \\
\qquad \mid \textbf{exception } \textit{Sys\_error } \_ \to \textit{read\_config\_in\_path filename dirs}
\end{array}
$$

Exceptions — Termination

Do all well-typed programs terminate in the presence of exceptions?

No, because exceptions hide the type of values that they communicate to the handler, which can be used to emulate recursive types.

Encode values of type τ_0 as lazy values of type *unit* $\rightarrow \tau_0$, say τ
Let encode be `fun x () -> x` and decode be `fun x -> x ()`.
Let dummy be some value of type τ_0.
Let type *exn* be $\tau \rightarrow \tau$, say σ.

Define the two coercion functions between types σ and τ:

$$\textit{fold} : \sigma \rightarrow \tau \triangleq \lambda f{:}\sigma.\, \lambda().\ \textit{let}\ \_ = \textit{raise}\ f\ \textit{in}\ \text{dummy}$$

$$\textit{unfold} : \tau \rightarrow \sigma \triangleq \lambda f{:}\tau.\ \textit{try}\ \textit{let}\ \_ = f\,()\ \textit{in}\ \lambda x{:}\tau.\, x\ \textit{with}\ \lambda y{:}\tau \rightarrow \tau.\, y$$

We may then define $\omega \triangleq \lambda x.\, (\textit{unfold}\ x)\ x$ so that $\omega\ (\textit{fold}\ \omega)$ loops.

Or a call-by-value fixpoint of type $(\sigma \rightarrow \sigma) \rightarrow \sigma$ that allows recursive definition of functions of type $\tau \rightarrow \tau$ (encoding type $\tau_0 \rightarrow \tau_0$).

Exercise

Program factorial with the previous encoding without using recursion (nor recursive types, nor references)

Exercise — Semantics of *let* $\cdot$ = $\cdot$ *with* $\cdot$ *in*

Describes the dynamic semantics of the *let* x = M_1 *with* M_2 *in* M_3.

Solution

We need a new evaluation context:

$$E ::= \ldots \mid \textit{let}\ x = E\ \textit{with}\ M_2\ \textit{in}\ M_3$$

and the following reduction rules:

Raise
$$F[\textit{raise}\ V] \longrightarrow \textit{raise}\ V$$

Handle-Val
$$\textit{let}\ x = V\ \textit{with}\ M_2\ \textit{in}\ M_3 \longrightarrow [x \mapsto V]M_3$$

Handle-Raise
$$\textit{let}\ x = \textit{raise}\ V\ \textit{with}\ M_2\ \textit{in}\ M_3 \longrightarrow M_2\ V$$

Exercise — Try finalize

A finalizer is some code that should always be run, whether the evaluation ends normally or an exception is being raised.

Write a function try_finalize that takes four arguments f, x, g, and y and returns the application $f\ x$ with finalizing code $g\ y$. *i.e.* $g\ y$ should be called before returning the result of the application of f to x whether it executed normally or raised an exception.

(You may try first without using binding mixed with exceptions, then using it, and compare.)

Exercise (Solution to) Try finalize

Without *let* · = · *with* · *in* :

```
let finalize f x g y =
  let result = try f x with exn → g y; raise exn in g y; result
```

An alternative version that does not duplicate the finalizing code and could be inlined, but allocates an intermediate result, is:

```
type 'a result = Val of 'a | Exc of exn
let finalize f x g y =
  let result = try Val (f x) with exn → Exc exn in
  g y; match result with Val x → x | Exc exn → raise exn
```

More concisely:

```
let finalize f x g y =
  match f x with
  | result → g y; result
  | exception exn → g y; raise exn
```

Generalizing exceptions Effect handlers

Exceptions allow to abort the current computation to the dynamically enclosing handler.

Effect handlers are a variant of control operators.

As exceptions, they allow to abort the current computation to the dynamically enclosing handler, but offer the handler the possibility to resume the computation where it was aborted.

They are (much) more expressive.

They also allow to model a global state, where a toplevel heap handler is setup so that allocation, read, and write can be implemented by passing control to the handler together with the current continuation, *i.e.* evaluation context, which may change the heap and then resume or throw away the continuation.

Contents

References

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

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

Expressions and evaluation contexts are extended:

$$\begin{array}{rcl} M & ::= & \ldots \mid \textit{ref}\, M \mid M := M \mid !\, M \\ E & ::= & \ldots \mid \textit{ref}\, [] \mid [] := M \mid V := [] \mid !\, [] \end{array}$$

References

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

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

(which intuitively should yield ?) would reduce to:

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

(which yields 3).

How shall we solve this problem?

References

(*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.

References

We extend the simply-typed λ-calculus calculus with *memory locations*:

$$\begin{array}{rcl} V & ::= & \ldots \mid \ell \\ M & ::= & \ldots \mid \ell \end{array}$$

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.

References

A *configuration* is a pair $M \,/\, \mu$ of a term and a store. The operational semantics (given next) reduces configurations instead of expressions.

The semantics maintains a no-dangling-pointers invariant: the locations that appear in M 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.
- If we wish to start reduction with a non-empty store, we must check that the initial configuration satisfies the *no-dangling-pointers* invariant.

References

Because the semantics now reduces configurations, all existing reduction rules are augmented with a store, which they do not touch:

$$(\lambda x{:}\tau.\,M)\;V\;/\;\mu \longrightarrow [x \mapsto V]M\;/\;\mu$$

$$E[M]\;/\;\mu \longrightarrow E[M']\;/\;\boxed{\mu'} \qquad \text{if } M\;/\;\mu \longrightarrow M'\;/\;\boxed{\mu'}$$

Three new reduction rules are added:

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

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

References

The type system is modified as follows. Types are extended:

$$\tau ::= \ldots \mid \mathit{ref}\,\tau$$

Three new typing rules are introduced:

$$\textsc{Ref}\quad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathit{ref}\,M : \mathit{ref}\,\tau} \qquad \textsc{Set}\quad \frac{\Gamma \vdash M_1 : \mathit{ref}\,\tau \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 := M_2 : \mathit{unit}} \qquad \textsc{Get}\quad \frac{\Gamma \vdash M : \mathit{ref}\,\tau}{\Gamma \vdash\, !\,M : \tau}$$

Is that all we need?

References

The preceding setup is enough to typecheck *source terms*, but does 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 M / μ well-typed?

References

When does a location ℓ have type *ref* τ?

A possible answer is, *when it points to some value of type τ*.
Intuitively, this could be formalized by a typing rule of the form:

$$\frac{\mu, \varnothing \vdash \mu(\ell) : \tau}{\mu, \Gamma \vdash \ell : \mathit{ref}\,\tau}$$

Comments?

- Typing judgments would have the form $\mu, \Gamma \vdash M : \tau$. However, they would no longer be *inductively* defined (or else, every cyclic structure would be ill-typed). Instead, *co-induction* would be required.
- Moreover, if the value $\mu(\ell)$ happens to admit two distinct types τ_1 and τ_2, then ℓ admits types *ref* τ_1 and *ref* τ_2. So, one can write at type τ_1 and read at type τ_2: this rule is *unsound!*

References

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* Σ, a finite mapping of locations to types.

So, when does a location ℓ have type *ref* τ? "When Σ says so."

$$\frac{\textsc{Loc}}{\Sigma, \Gamma \vdash \ell : \mathit{ref}\, \Sigma(\ell)}$$

Comments:

- Typing judgments now have the form $\Sigma, \Gamma \vdash M : \tau$.

References

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

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

$$\textsc{Store}\quad \frac{\forall \ell \in \mathrm{dom}(\mu), \quad \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vdash \mu : \Sigma} \qquad \textsc{Config}\quad \frac{\vdash \mu : \Sigma \qquad \Sigma, \varnothing \vdash M : \tau}{\vdash M \,/\, \mu : \tau}$$

Comments:

- This is an *inductive* definition. The store typing Σ serves both as an assumption (Loc) and a goal (Store). Cyclic stores are not a problem.
- The store typing is used only in the definition of a “well-typed configuration” and in the typechecking of locations. Thus, it is not needed for type-checking source programs, since the store is empty and the empty-store configuration is always well-typed.
- Notice that Σ does not appear in the conclusion of Config.

Restating type soundness

The type soundness statements are slightly modified in the presence of the store, since we now reduce configurations:

Theorem (Subject reduction)

Reduction preserves types: if $M \,/\, \mu \longrightarrow M' \,/\, \mu'$ *and* $\vdash M \,/\, \mu : \tau$*, then* $\vdash M' \,/\, \mu' : \tau$*.*

Theorem (Progress)

If $M \,/\, \mu$ *is a well-typed, irreducible configuration, then* M *is a value.*

Restating subject reduction

Inlining CONFIG, subject reduction can also be restated as:

Theorem (Subject reduction, expanded)

If $M \,/\, \mu \longrightarrow M' \,/\, \mu'$ and $\vdash \mu : \Sigma$ and $\Sigma, \varnothing \vdash M : \tau$, then there exists Σ' such that $\vdash \mu' : \Sigma'$ and $\Sigma', \varnothing \vdash M' : \tau$.

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

Establishing subject reduction

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

The hypotheses are:

$$M \,/\, \mu \longrightarrow M' \,/\, \mu' \quad \text{and} \quad \vdash \mu : \Sigma \quad \text{and} \quad \Sigma, \varnothing \vdash E[M] : \tau$$

Assuming compositionality, there exists τ' such that:

$$\Sigma, \varnothing \vdash M : \tau' \quad \text{and} \quad \forall M', \quad (\Sigma, \varnothing \vdash M' : \tau') \Rightarrow (\Sigma, \varnothing \vdash E[M'] : \tau)$$

Then, by the induction hypothesis, there exists Σ' such that:

$$\vdash \mu' : \Sigma' \quad \text{and} \quad \Sigma', \varnothing \vdash M' : \tau'$$

Here, *we are stuck*. The context E is well-typed under Σ, but the term M' is well-typed under Σ', so we cannot combine them.

How can we fix this?

Establishing subject reduction

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)

If $M \,/\, \mu \longrightarrow M' \,/\, \mu'$ *and* $\vdash \mu : \Sigma$ *and* $\Sigma, \varnothing \vdash M : \tau$*, then there exists* Σ' *such that* $\vdash \mu' : \Sigma'$ *and* $\Sigma', \varnothing \vdash M' : \tau$ *and* $\Sigma \subseteq \Sigma'$*.*

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

Establishing subject reduction

Growing the store typing preserves well-typedness:

Lemma (Stability under memory allocation)

If $\Sigma \subseteq \Sigma'$ *and* $\Sigma, \Gamma \vdash M : \tau$*, then* $\Sigma', \Gamma \vdash M : \tau$*.*

(This is a generalization of the weakening lemma.)

Establishing subject reduction

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

Lemma (Compositionality)

Assume $\Sigma, \varnothing \vdash E[M] : \tau$. *Then, there exists* τ' *such that:*

- $\Sigma, \varnothing \vdash M : \tau'$,
- *for every* Σ' *such that* $\Sigma \subseteq \Sigma'$, *for every* M', $\Sigma', \varnothing \vdash M' : \tau'$ *implies* $\Sigma', \varnothing \vdash E[M'] : \tau$.

Establishing subject reduction

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

The hypotheses are:

$$\vdash \mu : \Sigma \quad \text{and} \quad \Sigma, \varnothing \vdash E[M] : \tau \quad \text{and} \quad M \,/\, \mu \longrightarrow M' \,/\, \mu'$$

By compositionality, there exists τ' such that:

$$\Sigma, \varnothing \vdash M : \tau'$$
$$\forall \Sigma', \forall M', \quad (\Sigma \subseteq \Sigma') \Rightarrow (\Sigma', \varnothing \vdash M' : \tau') \Rightarrow (\Sigma', \varnothing \vdash E[M'] : \tau')$$

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

$$\vdash \mu' : \Sigma' \quad \text{and} \quad \Sigma', \varnothing \vdash M' : \tau' \quad \text{and} \quad \Sigma \subseteq \Sigma'$$

The goal immediately follows.

Exercise

Exercise (Recommended)

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

Monads

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

$$
\begin{aligned}
\text{return} :&\quad \alpha \to \mathit{IO}\,\alpha \\
\text{bind} :&\quad \mathit{IO}\,\alpha \to (\alpha \to \mathit{IO}\,\beta) \to \mathit{IO}\,\beta \\
\text{main} :&\quad \mathit{IO}\,() \\
\text{newIORef} :&\quad \alpha \to \mathit{IO}\,(\mathit{IORef}\,\alpha) \\
\text{readIORef} :&\quad \mathit{IORef}\,\alpha \to \mathit{IO}\,\alpha \\
\text{writeIORef} :&\quad \mathit{IORef}\,\alpha \to \alpha \to \mathit{IO}\,()
\end{aligned}
$$

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 [2008] for citations.

See also the Mezzo language [Pottier and Protzenko, 2013] designed especially for the explicit control of resources.

A similar approach is taken in the language Rust.

Contents

Combining extensions

We have shown how to extend simply-typed λ-calculus, independently, with:

- polymorphism, and
- references.

Can these two extensions be combined?

Beware of polymorphic locations!

When adding references, we noted that type soundness relies on the fact that *every reference cell (or memory location) has a fixed type*.

Otherwise, if a location had two types $\mathit{ref}\,\tau_1$ and $\mathit{ref}\,\tau_2$, one could store a value of type τ_1 and read back a value of type τ_2.

Hence, it should also be *unsound if a location could have type $\forall\alpha.\,\mathit{ref}\,\tau$* (where α appears in τ) as it could then be specialized to both types $\mathit{ref}\,([\alpha \mapsto \tau_1]\tau)$ and $\mathit{ref}\,([\alpha \mapsto \tau_2]\tau)$.

By contrast, *a location ℓ can have type $\mathit{ref}\,(\forall\alpha.\,\tau)$*: this says that ℓ stores values of polymorphic type $\forall\alpha.\,\tau$, but ℓ, as a value, is viewed with the monomorphic type $\mathit{ref}\,(\forall\alpha.\,\tau)$.

A counter example

Still, if naively extended with references, System F allows construction of polymorphic references, which breaks subject reduction:

$$\begin{array}{l} \mathit{let}\ y : \forall\alpha.\, \mathit{ref}\,(\alpha \to \alpha) = \Lambda\alpha.\, \mathit{ref}\,(\alpha \to \alpha)\ (\lambda z{:}\alpha.\, z)\ \mathit{in} \\ \quad (y\ \mathit{bool}) := (\mathit{bool} \to \mathit{bool})\ not; \\ \quad !(\mathit{int} \to \mathit{int})\ (y\ \mathit{int})\ 1 \,/\, \varnothing \\ \xrightarrow{*}\ not\ 1 \,/\, \ell \mapsto not \end{array}$$

What happens is that the evaluation of the reference:

- creates and returns a location ℓ bound to the identity function $\lambda z{:}\alpha.\, z$ of type $\alpha \to \alpha$,
- abstracts α in the result and binds it to y with the polymorphic type $\forall\alpha.\, \mathit{ref}\,(\alpha \to \alpha)$;
- writes the location at type $\mathit{ref}\,(\mathit{bool} \to \mathit{bool})$ and reads it back at type $\mathit{ref}\,(\mathit{int} \to \mathit{int})$.

Nailing the bug

In the counter-example, the first reduction step uses the following rule (where V is $\lambda x{:}\alpha.\,x$ and τ is $\alpha \to \alpha$).

$$\textsc{Context}\ \frac{\mathit{ref}\,\tau\ V\ /\ \varnothing \longrightarrow \ell\ /\ \ell \mapsto V}{\Lambda\alpha.\,\mathit{ref}\,\tau\ V\ /\ \varnothing \longrightarrow \Lambda\alpha.\,\ell\ /\ \ell \mapsto V}$$

While we have

$$\alpha \vdash \mathit{ref}\,\tau\ V\ /\ \varnothing : \mathit{ref}\,\tau \qquad \text{and} \qquad \alpha \vdash \ell\ /\ \ell \mapsto V : \mathit{ref}\,\tau$$

We have

$$\vdash \Lambda\alpha.\,\mathit{ref}\,\tau\ V\ /\ \varnothing : \forall\alpha.\,\mathit{ref}\,\tau \qquad \text{but not} \qquad \vdash \Lambda\alpha.\,\ell\ /\ \ell \mapsto V : \forall\alpha.\,\mathit{ref}\,\tau$$

Hence, the context case of subject reduction breaks.

Nailing the bug

The typing derivation of $\Lambda\alpha.\,\ell$ requires a store typing Σ of the form $\ell : \tau$ and a derivation of the form:

$$\textsc{Tabs}\ \frac{\Sigma, \alpha \vdash \ell : \mathit{ref}\,\tau}{\Sigma \vdash \Lambda\alpha.\,\ell : \forall\alpha.\,\mathit{ref}\,\tau}$$

However, the typing context Σ, α is ill-formed as α appears free in Σ.

Instead, a well-formed premise should bind α earlier as in $\alpha, \Sigma \vdash \ell : \mathit{ref}\,\tau$, but then, Rule Tabs cannot be applied.

By contrast, the expression $\mathit{ref}\,\tau\ V$ is pure, so Σ may be empty:

$$\textsc{Tabs}\ \frac{\alpha \vdash \mathit{ref}\,\tau\ V : \mathit{ref}\,\tau}{\varnothing \vdash \Lambda\alpha.\,\mathit{ref}\,\tau\ V : \forall\alpha.\,\mathit{ref}\,\tau}$$

The expression $\Lambda\alpha.\,\ell$ is correctly rejected as ill-typed, so $\Lambda\alpha.\,(\mathit{ref}\,\tau\ V)$ should also be rejected.

Fixing the bug

Mysterious slogan:

> *One must not abstract over a type variable that might, after evaluation of the term, enter the store typing.*

Indeed, this is what happens in our example. The type variable α which appears in the type $\alpha \to \alpha$ of V is abstracted in front of *ref* $(\alpha \to \alpha)\ V$.

When *ref* $(\alpha \to \alpha)\ V$ reduces, $\alpha \to \alpha$ becomes the type of the fresh location ℓ, which appears in the new store typing.

This is all well and good, but *how* do we enforce this slogan?

Fixing the bug

In the context of ML, a number of rather complex historic approaches have been followed: see Leroy [1992] for a survey.

Then came Wright [1995], who suggested an amazingly simple solution, known as the *value restriction:* only value forms can be abstracted over.

$$\text{TAbs}\quad \frac{\Gamma, \alpha \vdash u : \tau}{\Gamma \vdash \Lambda\alpha.\, u : \forall\alpha.\tau}$$

Value forms:

$$u ::= x \mid V \mid \Lambda\alpha.\, u \mid u\ \tau$$

The problematic proof case *vanishes*, as we now never $\beta\delta$-reduce under type abstraction—only ι-reduction is allowed.

Subject reduction holds again.

A good intuition: internalizing configurations

A configuration $M \,/\, \mu$ is an expression M in a memory μ. The memory can be viewed as a recursive extensible record.

The configuration $M \,/\, \mu$ may be viewed as the recursive definition (of values) $\textit{let rec } m : \Sigma = \mu \textit{ in } [\ell \mapsto m.\ell]M$ where Σ is a store typing for μ.

The store typing rules are coherent with this view.

Allocation of a reference is a reduction of the form

$$\begin{array}{rl} & \textit{let rec } m : \Sigma = \mu \textit{ in } E[\textit{ref}\,\tau\ V] \\ \longrightarrow & \textit{let rec } m : \Sigma, \ell : \tau = \mu, \ell \mapsto V \textit{ in } E[m.\ell] \end{array}$$

For this transformation to preserve well-typedness, it is clear that the evaluation context E *must not bind any free type variable of* τ.

Otherwise, we are violating the scoping rules.

Clarifying the typing rules

Let us review the typing rules for configurations:

$$\textsc{Config}\quad \frac{\vec{\alpha}, \Sigma, \varnothing \vdash M : \tau \qquad \vec{\alpha} \vdash \mu : \Sigma}{\vec{\alpha} \vdash M \,/\, \mu : \tau} \qquad\qquad \textsc{Store}\quad \frac{\forall \ell \in \mathrm{dom}(\mu), \quad \vec{\alpha}, \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vec{\alpha} \vdash \mu : \Sigma}$$

Remarks:

- Closed configurations are typed in an environment just composed of type variables $\vec{\alpha}$.
- $\vec{\alpha}$ may appear in the store during reduction.
  Take for example, M equal to *ref* $(\alpha \to \alpha)\ V$ where V is $\lambda x{:}\alpha.\, x$.
- Thus $\vec{\alpha}$ will also appear in the store typing and should be placed in front of the store typing; no β in $\vec{\alpha}$ can be generalized.
- New type variables cannot be introduced during reduction.

Clarifying the typing rules

Judgments are now of the form $\vec{\alpha}, \Sigma, \Gamma \vdash M : \tau$ although we may see $\vec{\alpha}, \Sigma, \Gamma$ as a whole typing context Γ'.

For locations, we need a new context formation rule:

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \tau \qquad \ell \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \ell : \tau}\ \textsc{WfEnvLoc}$$

This allows locations to appear anywhere. However, in a derivation of a closed term, the typing context will always be of the form $\vec{\alpha}, \Sigma, \Gamma$ where:

- Σ only binds locations (to arbitrary types) and
- Γ does not bind locations.

Clarifying the typing rules

The typing rule for memory locations (where Γ is of the form $\vec{\alpha}, \Sigma, \Gamma'$)

$$\frac{\textsc{Loc}}{\Gamma \vdash \ell : \textit{ref}\, \Gamma(\ell)}$$

In System F, typing rules for references need not be primitive.
We may instead treat them as constants of the following types:

$$\begin{array}{rcl} \textit{ref} & : & \forall \alpha.\, \alpha \to \textit{ref}\, \alpha \\ (!) & : & \forall \alpha.\, \textit{ref}\, \alpha \to \alpha \\ (:=) & : & \forall \alpha.\, \textit{ref}\, \alpha \to \alpha \to \textit{unit} \end{array}$$

There are all destructors (event *ref*) with the obvious arities.

The δ-rules are adapted to carry explicit type parameters:

$$\begin{array}{rcll} \textit{ref}\, \tau\ V \,/\, \mu & \longrightarrow & \ell \,/\, \mu[\ell \mapsto V] & \text{if } \ell \notin \mathrm{dom}(\mu) \\ \ell := (\tau)\ \ V \,/\, \mu & \longrightarrow & () \,/\, \mu[\ell \mapsto V] & \\ !\tau\ \ell \,/\, \mu & \longrightarrow & \mu(\ell) \,/\, \mu & \end{array}$$

Stating type soundness

Lemma (Subject reduction for constants)

δ-rules preserve well-typedness of closed configurations.

Theorem (Subject reduction)

Reduction of closed configurations preserves well-typedness.

Lemma (Progress for constants)

A well-typed closed configuration M/μ where M is a full application of constants ref, $(!)$, or $(:=)$ to types and values can always be reduced.

Theorem (Progress)

A well-typed irreducible closed configuration M/μ is a value.

Consequences

The problematic program is now syntactically ill-formed:

$$\begin{array}{l}\mathit{let}\; y : \forall\alpha.\, \mathit{ref}\,(\alpha \to \alpha) = \Lambda\alpha.\, \mathit{ref}\,(\lambda z{:}\alpha.\, z)\;\mathit{in}\\ \quad (:=)\;(\mathit{bool} \to \mathit{bool})\;(y\;\mathit{bool})\;not;\\ \quad !\;(\mathit{int} \to \mathit{int})\;(y\;(\mathit{int}))\;1\end{array}$$

Indeed, $\mathit{ref}\,(\lambda z{:}\alpha.\, z)$ is not a value form, but the application of a unary destructor to a value, so it cannot be generalized.

Value restriction limitations

With the value restriction, some pure programs become ill-typed, even though they were well-typed in the absence of references.

Therefore, this style of introducing references in System F (or in ML) is *not a conservative extension.*

Assuming:

$$map : \forall\alpha.\forall\beta.(\alpha \to \beta) \to \mathit{list}\,\alpha \to \mathit{list}\,\beta \qquad id : \forall\alpha.\alpha \to \alpha$$

This expression becomes ill-typed:

$$\Lambda\alpha.\,map\;\alpha\;\alpha\;(id\;\alpha)$$

A common work-around is to perform a manual η-expansion:

$$\Lambda\alpha.\,\lambda y{:}\mathit{list}\,\alpha.\,map\;\alpha\;(id\;\alpha)\;y$$

Of course, in the presence of side effects, η-expansion is *not* semantics-preserving, so this must not be done blindly.

Value restriction

Non-expansive expressions

The value restriction can be slightly relaxed by enlarging the class of value-forms to a syntactic category of so-called *non-expansive terms*—terms whose evaluation will definitely not allocate new reference cells. Non-expansive terms form a strict superset of value-forms.

$$
\begin{array}{rcl}
u & ::= & x \mid V \mid \Lambda\alpha.\, u \mid u\,\tau \\
& \mid & \mathit{let}\ x = u\ \mathit{in}\ u \mid (\lambda x{:}\tau.\, u)\ u \\
& \mid & C\ u_1\ \ldots\ u_k \\
& \mid & d\ u_1\ \ldots u_k
\end{array}
\qquad \text{where either} \left[\begin{array}{l} k < \mathit{arity}\,(d) \\ d \text{ is non-expansive.} \end{array}\right.
$$

In particular, pattern matching is a non-expansive destructor! But $\mathit{ref}\,\cdot$ is an expansive one!.

For example, the following expression is non-exapnsive:

$$\Lambda\alpha.\, \mathit{let}\ x = (\mathit{match}\ y\ \mathit{with}\ (C_i\ \bar{x}_i \to u_i)^{\,i\in I}))\ \mathit{in}\ u$$

Value restriction

Positive occurrences: Garrigue [2004] relaxes the value restriction in a more subtle way, which is justified by a subtyping argument.

For instance, *let* $x : \forall\alpha.\,$*list* $\alpha = \Lambda\alpha.\,(M_1\ M_2)$ *in* M may be well-typed because because α appears only positively in the type of $M_1\ M_2$.

More generally, given a type context $T[\alpha]$ where α appears only positively

- $\forall\alpha.\,T[\alpha]$ can be instantiated to $T[\forall\alpha.\,\alpha]$, and
- $T[\forall\alpha.\,\alpha]$ is a subtype of $\forall\alpha.\,T[\alpha]$

Hence, a value of type $T[\alpha]$ can be given the monomorphic type $T[\forall\alpha.\,\alpha]$ by weakening before entering the store to please the value restriction, but retrieved at type $\forall\alpha.\,T[\alpha]$, a subtype of $T[\forall\alpha.\,\alpha]$.

OCaml implements this, but restricts it to *strictly positive* occurrences so as to keep the principal type property.

Value restriction

Extensions

In fact, the two extensions can be combined: $\Lambda\alpha.\,M$ *need only be forbidden* when

> α *appears in the type of some exposed expansive subterm at some negative occurrence,*

where exposed subterms are those that do not appear under some λ-abstraction.

For instance, the expression

$$
\begin{array}{l}
\textit{let}\ x : \forall\alpha.\,\textit{int} \times (\textit{list}\ \alpha) \times (\alpha \to \alpha) = \\
\quad \Lambda\alpha.\,(\textit{ref}\,(1+2),\ (\lambda x{:}\alpha.\,x)\ \textit{Nil},\ \lambda x{:}\alpha.\,x) \\
\textit{in}\ M
\end{array}
$$

may be accepted because α appears only in the type of the non-expansive exposed expression $\lambda x{:}\alpha.\,x$ and only positively in the type of the expansive expression $(\lambda x{:}\alpha.\,x)$ *Nil*.

Conclusions

Experience has shown that *the value restriction is tolerable.* Even though it is not conservative, the search for better solutions has been pretty much abandoned.

There is still on going research for tracing side effects more precisely, in particular to better circumvent their use.

Actually, there is a regained interest in tracing side effects, with the introduction of effect handlers.

Conclusions

In a type-and-effect system [Lucassen and Gifford, 1988; Talpin and Jouvelot, 1994], or in a type-and-capability system [Charguéraud and Pottier, 2008], the type system indicates which expressions may allocate new references, and at which type. This permits strong updates—updates that may also change the type of references.

There, the value restriction is no longer necessary.

However, if one extends a type-and-capability system with a mechanism for *hiding* state, the need for the value restriction re-appears.

Pottier and Protzenko [2012] (and [Protzenko, 2014]) designed a language, called Mezzo, where mutable state is tracked very precisely, using permissions, ownership, and afine types.

Type reconstruction

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?

Logical versus algorithmic properties

We have seen three different type systems, simply-typed λ-calculus, ML, and System F, of increasing expressiveness.

In each case, we have presented an explicitly-typed and an implicitly-typed version of the language and shown a close correspondence between the two views, thanks to a type-passing semantics.

We argued that the explicitly-typed version is often more convenient for studying the meta-theoretical properties of the language.

Which one should we used for checking well-typedness? That is, in which language should we write programs?

Checking type derivations

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

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 usually 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.

Bottom-up type-checking

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

One seeks an *algorithmic reading* of the typing rules, where, in a judgment $\Gamma \vdash M : \tau$, the parameters Γ and M are *inputs*, while the parameter τ is an *output*.

Moreover, typing rules should be such that a type appearing as output in a conclusion should also appear as output in a premise or as input in the conclusion and input in the premises should be input of the conclusion or output of other premises.

$$\textsc{Abs — Checking rule}\quad \frac{\Gamma, x : \tau_0^{\uparrow} \vdash M : \tau^{\downarrow}}{\Gamma \vdash \lambda x{:}\tau_0^{\uparrow}.\, M : \tau_0^{\downarrow} \rightarrow \tau^{\downarrow}} \qquad \textsc{Abs — Inference rule}\quad \frac{\Gamma, x : \tau_0^{\uparrow} \vdash a : \tau^{\downarrow}}{\Gamma \vdash \lambda x.\, a : \tau_0^{\downarrow} \rightarrow \tau^{\downarrow}}$$

This way, types need never be guessed, just looked up into the typing context, instantiated, or checked for equality.

Bottom-up type-checking

This is exactly the situation with explicitly-typed presentations of the typing rules.

This is also 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.

Bottom-up type-checking

However, this implies a lot of redundancies:

- Parameters of *all* functions need to be annotated, even when their types are obvious from context.
- Let-expressions (when not primitive), recursive definitions, injections into sum types need to be annotated.
- As the language grows, more and more constructs require type annotations, *e.g.* type applications and type abstractions.

Type annotations may quickly obfuscate the code and large explicitly-typed terms are so verbose that they become intractable by humans!

Hence, programming in the implicitly-typed version is more appealing.

Type inference

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

For System F, this is however undecidable. Since programming in explicitly-typed System F is not practically feasible, some amount of type reconstruction must still be done. Typically, the algorithm is incomplete, *i.e.* it rejects terms that are perhaps well-typed, but the user may always provide more annotations and at worse, the explicitly-typed version is never rejected if well-typed.

Contents

Type inference

The type inference algorithm for simply-typed λ-calculus, is due to Hindley [1969]. The idea behind the 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.

Type inference

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*.

Constraints

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.

Constraints

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

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

A type is either a *type variable* α 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

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

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

Ground types contain no variables. The base case in this definition is when F has arity zero. *There should be at least one constructor of arity zero, so that the model is non-empty.*

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.

Constraints

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

This judgment is inductively defined:

$$\phi \vdash \mathit{true} \qquad \frac{\phi\tau_1 = \phi\tau_2}{\phi \vdash \tau_1 = \tau_2} \qquad \frac{\phi \vdash C_1 \qquad \phi \vdash C_2}{\phi \vdash C_1 \wedge C_2} \qquad \frac{\phi[\alpha \mapsto \mathbf{t}] \vdash C}{\phi \vdash \exists\alpha.C}$$

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

We 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*.

Constraint generation

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

$$
\begin{array}{rcl}
\langle\!\langle \Gamma \vdash x : \tau \rangle\!\rangle & = & \Gamma(x) = \tau \\
\langle\!\langle \Gamma \vdash \lambda x. a : \tau \rangle\!\rangle & = & \exists \alpha_1 \alpha_2.(\langle\!\langle \Gamma, x : \alpha_1 \vdash a : \alpha_2 \rangle\!\rangle \wedge \alpha_1 \rightarrow \alpha_2 = \tau) \\
 & & \text{if } \alpha_1, \alpha_2 \# \Gamma, a, \tau \\
\langle\!\langle \Gamma \vdash a_1\ a_2 : \tau \rangle\!\rangle & = & \exists \alpha.(\langle\!\langle \Gamma \vdash a_1 : \alpha \rightarrow \tau \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash a_2 : \alpha \rangle\!\rangle) \\
 & & \text{if } \alpha \# \Gamma, a_1, a_2, \tau
\end{array}
$$

Thanks to the use of existential quantification, the names that occur free in $\langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$ are a subset of those that occur free in Γ or τ.

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

An example

Let us perform type inference for the closed term

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

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

$$\langle\!\langle \varnothing \vdash \lambda fxy.(f\ x, f\ y) : \alpha_0 \rangle\!\rangle$$

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

Solving the constraint means to find all possible ground assignments for α_0 that satisfy the constraint.

Typically, this is done by transforming the constraint into successive equivalent constraints until some constraint that is obviously satisfiable and from which solutions may be directly read.

An example

$$
\begin{array}{cl}
 & \langle\!\langle \varnothing \vdash \lambda fxy.\,(f\ x, f\ y) : \alpha_0 \rangle\!\rangle \\[2ex]
= & \exists \alpha_1\alpha_2.\left(\begin{array}{l} \langle\!\langle f : \alpha_1 \vdash \lambda xy.\ \ldots : \alpha_2 \rangle\!\rangle \\ \alpha_1 \to \alpha_2 = \alpha_0 \end{array}\right) \\[2ex]
= & \exists \alpha_1\alpha_2.\left(\begin{array}{l} \exists \alpha_3\alpha_4.\left(\begin{array}{l} \langle\!\langle f : \alpha_1; x : \alpha_3 \vdash \lambda y.\ \ldots : \alpha_4 \rangle\!\rangle \\ \alpha_3 \to \alpha_4 = \alpha_2 \end{array}\right) \\ \alpha_1 \to \alpha_2 = \alpha_0 \end{array}\right) \\[2ex]
= & \exists \alpha_1\alpha_2.\left(\begin{array}{l} \exists \alpha_3\alpha_4.\left(\begin{array}{l} \exists \alpha_5\alpha_6.\left(\begin{array}{l} \langle\!\langle f : \alpha_1; x : \alpha_3; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_5 \to \alpha_6 = \alpha_4 \end{array}\right) \\ \alpha_3 \to \alpha_4 = \alpha_2 \end{array}\right) \\ \alpha_1 \to \alpha_2 = \alpha_0 \end{array}\right)
\end{array}
$$

We perform constraint generation for the 3 λ-abstractions.

An example

$$\exists \alpha_1\alpha_2.\left(\begin{array}{l} \exists \alpha_3\alpha_4.\left(\begin{array}{l} \exists \alpha_5\alpha_6.\left(\begin{array}{l} \langle\!\langle f : \alpha_1; x : \alpha_3; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_5 \to \alpha_6 = \alpha_4 \end{array}\right) \\ \alpha_3 \to \alpha_4 = \alpha_2 \end{array}\right) \\ \alpha_1 \to \alpha_2 = \alpha_0 \end{array}\right)$$

$$\equiv \quad \exists \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6.\left(\begin{array}{l} \langle\!\langle f : \alpha_1; x : \alpha_3; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_5 \to \alpha_6 = \alpha_4 \\ \alpha_3 \to \alpha_4 = \alpha_2 \\ \alpha_1 \to \alpha_2 = \alpha_0 \end{array}\right)$$

We hoist up existential quantifiers:

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

An example

$$\exists \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6.\left(\begin{array}{l}\langle\!\langle f:\alpha_1; x:\alpha_3; y:\alpha_5 \vdash (f\ x, f\ y):\alpha_6\rangle\!\rangle\\ \alpha_5 \to \alpha_6 = \alpha_4\\ \alpha_3 \to \alpha_4 = \alpha_2\\ \alpha_1 \to \alpha_2 = \alpha_0\end{array}\right)$$

$$\equiv \quad \exists \alpha_1\alpha_2\alpha_3\alpha_5\alpha_6.\left(\begin{array}{l}\langle\!\langle f:\alpha_1; x:\alpha_3; y:\alpha_5 \vdash (f\ x, f\ y):\alpha_6\rangle\!\rangle\\ \alpha_3 \to \alpha_5 \to \alpha_6 = \alpha_2\\ \alpha_1 \to \alpha_2 = \alpha_0\end{array}\right)$$

We eliminate type variables with defining equations:

$$\boxed{\exists \alpha.(C \wedge \alpha = \tau) \equiv [\alpha \mapsto \tau]C} \qquad \text{if } \alpha \# \tau$$

An example

$$\exists \alpha_1 \alpha_2 \alpha_3 \alpha_5 \alpha_6 . \left(\begin{array}{l} \langle\!\langle f : \alpha_1 ; x : \alpha_3 ; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_3 \to \alpha_5 \to \alpha_6 = \alpha_2 \\ \alpha_1 \to \alpha_2 = \alpha_0 \end{array} \right)$$

$$\equiv \quad \exists \alpha_1 \alpha_3 \alpha_5 \alpha_6 . \left(\begin{array}{l} \langle\!\langle f : \alpha_1 ; x : \alpha_3 ; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_6 = \alpha_0 \end{array} \right)$$

We have again eliminated a type variable (α_2) with a defining equation.

In the following, let Γ stand for $(f : \alpha_1 ; x : \alpha_3 ; y : \alpha_5)$.

An example

$$\exists \alpha_1 \alpha_3 \alpha_5 \alpha_6 . \left(\begin{array}{l} \langle\!\langle \Gamma \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_6 = \alpha_0 \end{array} \right)$$

$$\equiv \quad \exists \alpha_1 \alpha_3 \alpha_5 \alpha_6 \alpha_7 \alpha_8 . \left(\begin{array}{l} \langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash f\ y : \alpha_8 \rangle\!\rangle \\ \alpha_7 \times \alpha_8 = \alpha_6 \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_6 = \alpha_0 \end{array} \right)$$

$$\equiv \quad \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8 . \left(\begin{array}{l} \langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash f\ y : \alpha_8 \rangle\!\rangle \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0 \end{array} \right)$$

We have performed constraint generation for the pair, hoisted the resulting existential quantifiers, and eliminated a type variable (α_6).

Let us now focus on the first application...

An example

$$
\begin{array}{rl}
& \langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle \\
= & \exists \alpha_9. \left(\begin{array}{l} \langle\!\langle \Gamma \vdash f : \alpha_9 \to \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash x : \alpha_9 \rangle\!\rangle \end{array} \right) \\
\\
= & \exists \alpha_9. \left(\begin{array}{l} \alpha_1 = \alpha_9 \to \alpha_7 \\ \alpha_3 = \alpha_9 \end{array} \right) \\
\\
\equiv & \alpha_1 = \alpha_3 \to \alpha_7
\end{array}
$$

We perform constraint generation for the variables f and x, and eliminate a type variable (α_9).

Recall that Γ stands for $(f : \alpha_1; x : \alpha_3; y : \alpha_5)$.

Now, back to the big picture...

An example

$$\exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8 . \begin{pmatrix} \langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash f\ y : \alpha_8 \rangle\!\rangle \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0 \end{pmatrix}$$

$$\equiv \quad \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8 . \begin{pmatrix} \alpha_1 = \alpha_3 \to \alpha_7 \\ \langle\!\langle \Gamma \vdash f\ y : \alpha_8 \rangle\!\rangle \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0 \end{pmatrix}$$

$$\equiv \quad \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8 . \begin{pmatrix} \alpha_1 = \alpha_3 \to \alpha_7 \\ \alpha_1 = \alpha_5 \to \alpha_8 \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0 \end{pmatrix}$$

We apply a simplification under a context:

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

An example

$$\exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8 . \left(\begin{array}{l} \alpha_1 = \alpha_3 \to \alpha_7 \\ \alpha_1 = \alpha_5 \to \alpha_8 \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0 \end{array} \right)$$

$$\equiv \quad \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8 . \left(\begin{array}{l} \alpha_1 = \alpha_3 \to \alpha_7 \\ \alpha_3 = \alpha_5 \\ \alpha_7 = \alpha_8 \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0 \end{array} \right)$$

$$\equiv \quad \exists \alpha_3 \alpha_7 . \left(\; (\alpha_3 \to \alpha_7) \to \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7 = \alpha_0 \; \right)$$

We apply transitivity at α_1, structural decomposition, and eliminate three type variables (α_1, α_5, α_8).

We have now reached a *solved form*.

An example

We have checked the following equivalence:

$$
\begin{array}{ll}
 & \langle\!\langle \varnothing \vdash \lambda f x y.\,(f\ x, f\ y) : \alpha_0 \rangle\!\rangle \\
\equiv & \exists \alpha_3 \alpha_7.\big(\ (\alpha_3 \to \alpha_7) \to \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7 = \alpha_0\ \big)
\end{array}
$$

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

$(\alpha_3 \to \alpha_7) \to \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7$ is a *principal type* for $\lambda f x y.\,(f\ x, f\ y)$.

An example

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

$$\langle\!\langle f : \alpha_1 \vdash \lambda xy.(f\ x, f\ y) : \alpha_2 \rangle\!\rangle$$

We have already done so... with only a slight difference: α_1 and α_2 are now free, so they cannot be eliminated.

An example

One can check the following equivalence:

$$\begin{array}{rl} & \langle\!\langle f : \alpha_1 \vdash \lambda xy.\,(f\ x, f\ y) : \alpha_2 \rangle\!\rangle \\ \equiv & \exists \alpha_3 \alpha_7. \left(\begin{array}{l} \alpha_3 \to \alpha_7 = \alpha_1 \\ \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7 = \alpha_2 \end{array} \right) \end{array}$$

In other words, the ground *typings* of $\lambda xy.\,(f\ x, f\ y)$ are all ground pairs of the form[2]:

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

Remember that a typing is a pair of an environment and a type.

[2] If we restrict to contexts of domain $\{x\}$, the only free variable of the term.

Typings

Definition

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

The type inference problem is to determine whether a term a 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.)

Constraint generation

Theorem (Soundness and completeness)

$\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$ *if and only if* $\phi\Gamma \vdash a : \phi\tau$.

Proof.

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

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

Constraint generation

Corollary

Let $\mathrm{fv}(a) = \{x_1, \ldots, x_n\}$*, where* $n \geq 0$*. Let* $\alpha_0, \ldots, \alpha_n$ *be pairwise distinct type variables. Then, the ground typings of* a *are described by*

$$((x_i : \phi\alpha_i)_{i \in 1..n},\ \phi\alpha_0)$$

where ϕ *ranges over all solutions of* $\langle\!\langle (x_i : \alpha_i)_{i \in 1..n} \vdash a : \alpha_0 \rangle\!\rangle$*.*

Corollary

Let $\mathrm{fv}(a) = \varnothing$*. Then,* a *is well-typed if and only if* $\exists\alpha.\langle\!\langle \varnothing \vdash a : \alpha \rangle\!\rangle \equiv \mathit{true}$*.*

Constraint solving

A constraint solving algorithm is typically presented as a (non-deterministic) 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*.

First-order unification as constraint solving

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

$$U ::= \textit{true} \mid \textit{false} \mid \epsilon \mid U \wedge U \mid \exists \bar{\alpha}.U$$

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

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

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

$$
\begin{array}{rcll}
(\exists\bar{\alpha}.U_1) \wedge U_2 & \longrightarrow & \exists\bar{\alpha}.(U_1 \wedge U_2) & \text{(extrusion)} \\
 & & \text{if } \bar{\alpha} \# U_2 & \\
\alpha = \epsilon \wedge \alpha = \epsilon' & \longrightarrow & \alpha = \epsilon = \epsilon' & \text{(fusion)} \\
\mathsf{F}\,\vec{\alpha} = \mathsf{F}\,\vec{\tau} = \epsilon & \longrightarrow & \vec{\alpha} = \vec{\tau} \wedge \mathsf{F}\,\vec{\alpha} = \epsilon & \text{(decomposition)} \\
\mathsf{F}\,\tau_1 \,\ldots\, \tau_i \,\ldots\, \tau_n = \epsilon & \longrightarrow & \exists\alpha.(\alpha = \tau_i \wedge \mathsf{F}\,\tau_1 \,\ldots\, \alpha \,\ldots\, \tau_n = \epsilon) & \text{(naming)} \\
 & & \text{if } \tau_i \text{ is not a variable} \wedge \alpha \# \tau_1, \ldots, \tau_n, \epsilon & \\
\mathsf{F}\,\vec{\tau} = \mathsf{F}'\,\vec{\tau}' = \epsilon & \longrightarrow & \mathit{false} & \text{(clash)} \\
 & & \text{if } \mathsf{F} \neq \mathsf{F}' & \\
U & \longrightarrow & \mathit{false} & \text{(occurs check)} \\
 & & \text{if } U \text{ is cyclic} & \\
\mathcal{U}[\mathit{false}] & \longrightarrow & \mathit{false} & \text{(error propag.)}
\end{array}
$$

See [Pottier and Rémy, 2005, §10.6] for additional administrative rules.

The occurs check

α *dominates* β (with respect to U) iff U contains a multi-equation of the form $\alpha = \mathsf{F}\,\tau_1 \ldots \beta \ldots \tau_n = \ldots$

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

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

Remark: Cyclic constraints would become solvable if we allowed regular trees for ground terms.

Solved forms

A solved form is either *false* or $\exists\bar{\alpha}.U$, 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 that is not *false* is satisfiable – indeed, a solution is easily constructed by well-founded recursion over the domination relation.

Implementation

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.

Contents

Two presentations

Two presentations of type inference for Damas and Milner's type system are possible:

- one of Milner's classic algorithms [1978], $\mathcal{W}$ or $\mathcal{J}$; see Pottier's old course notes for details [Pottier, 2002, §3.3];
- a constraint-based presentation [Pottier and Rémy, 2005];

We favor the latter, but quickly review the former first.

Preliminaries

This algorithm expects a pair $\Gamma \vdash a$, produces a type τ, and uses two global variables, $\mathcal{V}$ and φ.

$\mathcal{V}$ is an infinite *fresh supply* of type variables:

$$\begin{array}{rcl} \textit{fresh} & = & \textit{do}\ \alpha \in \mathcal{V} \\ & & \textit{do}\ \mathcal{V} \leftarrow \mathcal{V} \setminus \{\alpha\} \\ & & \textit{return}\ \alpha \end{array}$$

φ is an *idempotent substitution* (of types for type variables), initially the identity.

The algorithm

Here is the algorithm in monadic style:

$$
\begin{array}{rcl}
\mathcal{J}(\Gamma \vdash x) & = & \textit{let } \forall\alpha_1 \ldots \alpha_n.\tau = \Gamma(x) \\
 & & \textit{do } \alpha'_1, \ldots, \alpha'_n = \textit{fresh}, \ldots, \textit{fresh} \\
 & & \textit{return } [\alpha_i \mapsto \alpha'_i]_{i=1}^n(\tau) \text{ -- \textit{take a fresh instance}} \\
\mathcal{J}(\Gamma \vdash \lambda x.\, a_1) & = & \textit{do } \alpha = \textit{fresh} \\
 & & \textit{do } \tau_1 = \mathcal{J}(\Gamma; x : \alpha \vdash a_1) \\
 & & \textit{return } \alpha \to \tau_1 \text{ -- \textit{form an arrow type}} \\
 & \ldots &
\end{array}
$$

The algorithm

$$
\begin{array}{rcl}
 & \dots & \\
\mathcal{J}(\Gamma \vdash a_1\ a_2) & = & \textit{do}\ \tau_1 = \mathcal{J}(\Gamma \vdash a_1) \\
 & & \textit{do}\ \tau_2 = \mathcal{J}(\Gamma \vdash a_2) \\
 & & \textit{do}\ \alpha = \textit{fresh} \\
 & & \textit{do}\ \varphi \leftarrow \mathrm{mgu}(\varphi(\tau_1) = \varphi(\tau_2 \rightarrow \alpha)) \circ \varphi \\
 & & \textit{return}\ \alpha\ \textit{– solve}\ \tau_1 = \tau_2 \rightarrow \alpha \\
\mathcal{J}(\Gamma \vdash \textit{let}\ x = a_1\ \textit{in}\ a_2) & = & \textit{do}\ \tau_1 = \mathcal{J}(\Gamma \vdash a_1) \\
 & & \textit{let}\ \sigma = \forall \setminus \mathrm{ftv}(\varphi(\Gamma)).\, \varphi(\tau_1)\ \textit{– generalize} \\
 & & \textit{return}\ \mathcal{J}(\Gamma; x : \sigma \vdash a_2)
\end{array}
$$

($\forall \setminus \bar{\alpha}.\, \tau$ quantifies over all type variables *other than* $\bar{\alpha}$.)

Some weaknesses

Algorithm $\mathcal{J}$ mixes *generation* and *solving* of equations. This lack of modularity leads to several weaknesses:

- proofs are more difficult;
- correctness and efficiency concerns are not clearly separated (if implemented literally, the algorithm is exponential in practice);
- adding new language constructs duplicates solving of equations;
- generalizations, such as the introduction of subtyping, are not easy.

Some weaknesses

Algorithm $\mathcal{J}$ works with *substitutions*, instead of *constraints*.

Substitutions are an approximation to solved forms for unification constraints.

Working with substitutions means using *most general unifiers*, *composition*, and *restriction*.

Working with constraints means using *equations*, *conjunction*, and *existential quantification*.

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Road map

Type inference for Damas and Milner's type system involves slightly more than first-order unification: there is also *generalization* and *instantiation* of type schemes.

So, the constraint language must be enriched.

We proceed in two steps:

- still within simply-typed λ-calculus, we present a variation of the constraint language;
- building on this variation, we introduce polymorphism.

A variation on constraints

How about letting the constraint solver, instead of the constraint generator, deal with *environment access* and *construction?*

Let's enrich the syntax of constraints:

$$C ::= \ldots \mid x = \tau \mid \textit{def}\ x : \tau\ \textit{in}\ C$$

The idea is to interpret constraints in such a way as to validate the equivalence law:

$$\boxed{\textit{def}\ x : \tau\ \textit{in}\ C \equiv [x \mapsto \tau]C}$$

The *def* form is an *explicit substitution* form.

A variation on constraints

More precisely, here is the new interpretation of constraints.

As before, a valuation ϕ maps type variables α to ground types.

In addition, a valuation x_1 maps term variables x to ground types.

The satisfaction judgment now takes the form $\phi, x_1 \vdash C$. The new rules of interest are:

$$\frac{x_1 x = \phi\tau}{\phi, x_1 \vdash x = \tau} \qquad \frac{\phi, x_1[x \mapsto \phi\tau] \vdash C}{\phi, x_1 \vdash \textit{def } x : \tau \textit{ in } C}$$

(All other rules are modified to just transport x_1.)

A variation on constraints

Constraint generation is now a mapping of an expression a and a type τ to a constraint $\langle\!\langle a : \tau \rangle\!\rangle$. There is no longer a need for the parameter Γ.

$$
\begin{array}{rcl}
\langle\!\langle x : \tau \rangle\!\rangle & = & x = \tau \\
\langle\!\langle \lambda x. a : \tau \rangle\!\rangle & = & \exists \alpha_1 \alpha_2 . (\textit{def } x : \alpha_1 \textit{ in } \langle\!\langle a : \alpha_2 \rangle\!\rangle \wedge \alpha_1 \to \alpha_2 = \tau) \\
 & & \text{if } \alpha_1, \alpha_2 \# a, \tau \\
\langle\!\langle a_1\ a_2 : \tau \rangle\!\rangle & = & \exists \alpha . (\langle\!\langle a_1 : \alpha \to \tau \rangle\!\rangle \wedge \langle\!\langle a_2 : \alpha \rangle\!\rangle) \\
 & & \text{if } \alpha \# a_1, a_2, \tau
\end{array}
$$

No environments!

A variation on constraints

Theorem (Soundness and completeness)

Assume $\mathrm{fv}(a) = \mathrm{dom}(\Gamma)$. *Then,* $\phi, \phi\Gamma \vdash \langle\!\langle a : \tau \rangle\!\rangle$ *if and only if* $\phi\Gamma \vdash a : \phi\tau$.

Corollary

Assume $\mathrm{fv}(a) = \varnothing$. *Then,* a *is well-typed if and only if* $\exists\alpha.\langle\!\langle a : \alpha \rangle\!\rangle \equiv$ *true.*

Summary

This variation shows that there is *freedom* in the design of the constraint language, and that altering this design can *shift work* from the constraint generator to the constraint solver, or vice-versa.

Enriching constraints

To permit polymorphism, we must extend the syntax of constraints so that a variable x denotes not just a ground type, but a *set of ground types*.

However, these sets *cannot* be represented as type schemes $\forall\bar{\alpha}.\tau$, because constructing these simplified forms requires constraint solving.

To avoid mingling constraint generation and constraint solving, we use type schemes that incorporate constraints: *constrained type schemes*.

Enriching constraints

The syntax of *constraints* and of *constrained type schemes* is:

$$
\begin{array}{rcl}
C & ::= & \tau = \tau \mid C \wedge C \mid \exists \alpha . C \\
 & \mid & x \preceq \tau \\
 & \mid & \sigma \preceq \tau \\
 & \mid & \mathit{def}\ x : \sigma\ \mathit{in}\ C \\
\sigma & ::= & \forall \bar{\alpha}[C] . \tau
\end{array}
$$

$x \preceq \tau$ and $\sigma \preceq \tau$ are *instantiation constraints.*

$\sigma \preceq \tau$ constraints are introduced so as to make the syntax stable under substitutions of constrained type schemes for variables.

As before, *def* $x : \sigma$ *in* C is an *explicit substitution* form.

Enriching constraints

The idea is to interpret constraints in such a way as to validate the equivalence laws:

$$\mathit{def}\ x : \sigma\ \mathit{in}\ C \equiv [x \mapsto \sigma]C$$

$$(\forall \bar{\alpha}[C].\tau) \leq \tau' \equiv \exists \bar{\alpha}.(C \wedge \tau = \tau') \quad \text{if } \bar{\alpha} \# \tau'$$

Using these laws, a closed constraint can be rewritten to a unification constraint (with a possibly exponential increase in size).

The new constructs do not add much expressive power. They add just enough to allow a stand-alone formulation of constraint generation.

Interpreting constraints

A type variable α still denotes a ground type.

A variable x now denotes a *set* of ground types.

Instantiation constraints are interpreted as *set membership.*

$$\frac{\phi\tau \in x_1 x}{\phi, x_1 \vdash x \leq \tau} \qquad \frac{\phi\tau \in \binom{\phi}{x_1}\sigma}{\phi, x_1 \vdash \sigma \leq \tau} \qquad \frac{\phi, x_1[x \mapsto \binom{\phi}{x_1}\sigma] \vdash C}{\phi, x_1 \vdash \textit{def } x : \sigma \textit{ in } C}$$

Interpreting constrained type schemes

The interpretation of $\forall\bar\alpha[C].\tau$ under ϕ and x_1 is the set of all $\phi'\tau$, where ϕ and ϕ' coincide outside $\bar\alpha$ and where ϕ' and x_1 satisfy C.

$$\binom{\phi}{x_1}(\forall\bar\alpha[C].\tau) \;=\; \{\phi'\tau \mid (\phi' \setminus \bar\alpha = \phi \setminus \bar\alpha) \wedge (\phi', x_1 \vdash C)\}$$

For instance, the interpretation of $\forall\alpha[\exists\beta.\alpha = \beta \to \delta].\alpha \to \alpha$ under ϕ and x_1 is the set of all ground types of the form $(t \to \phi\delta) \to (t \to \phi\delta)$, where t ranges over ground types.

This is also the interpretation of $\forall\beta.(\beta \to \delta) \to (\beta \to \delta)$.

In fact, every constrained type scheme is equivalent to a standard type scheme. (Because constrainted can be reduced to equality constrained, which can always be eliminated: this would no longer be true if we introducted subtyping constrained.)

If $\bar\alpha$ and C are empty, then $\binom{\phi}{x_1}\tau$ is $\phi\tau$.

A derived form

Notice that *def* $x : \sigma$ *in* C is equivalent to C whenever x does not appear free in C—whether or not of the constraints appearing in σ are solvable.

To enforce the constraints in σ to be solvable, we use a variant of the *def* construct:

$$\textit{let}\ x : \sigma\ \textit{in}\ C \quad \equiv \quad \textit{def}\ x : \sigma\ \textit{in}\ ((\exists\alpha.x \leq \alpha) \wedge C)$$

Expanding $\sigma \triangleq \forall\bar{\alpha}[C_0].\tau$ and simplifying, an equivalent definition is:

$$\textit{let}\ x : \forall\bar{\alpha}[C_0].\tau\ \textit{in}\ C \quad \equiv \quad \exists\bar{\alpha}.C_0 \wedge \textit{def}\ x : \forall\bar{\alpha}[C_0].\tau\ \textit{in}\ C$$

It would also be equivalent to provide a direct interpretation of it:

$$\frac{(\begin{smallmatrix}\phi\\ x_1\end{smallmatrix})\sigma \neq \varnothing \qquad \phi, x_1[x \mapsto (\begin{smallmatrix}\phi\\ x_1\end{smallmatrix})\sigma] \vdash C}{\phi, x_1 \vdash \textit{let}\ x : \sigma\ \textit{in}\ C}$$

Constraint generation

Constraint generation is now as follows:

$$
\begin{array}{rcl}
\langle\!\langle x : \tau \rangle\!\rangle & = & x \preceq \tau \\
\langle\!\langle \lambda x. a : \tau \rangle\!\rangle & = & \exists \alpha_1 \alpha_2.(\textit{def}\ x : \alpha_1\ \textit{in}\ \langle\!\langle a : \alpha_2 \rangle\!\rangle \wedge \alpha_1 \to \alpha_2 = \tau) \\
 & & \text{if } \alpha_1, \alpha_2 \# a, \tau \\
\langle\!\langle a_1\ a_2 : \tau \rangle\!\rangle & = & \exists \alpha.(\langle\!\langle a_1 : \alpha \to \tau \rangle\!\rangle \wedge \langle\!\langle a_2 : \alpha \rangle\!\rangle) \\
 & & \text{if } \alpha \# a_1, a_2, \tau \\
\langle\!\langle \textit{let}\ x = a_1\ \textit{in}\ a_2 : \tau \rangle\!\rangle & = & \textit{let}\ x : (\!| a_1 |\!)\ \textit{in}\ \langle\!\langle a_2 : \tau \rangle\!\rangle \\
(\!| a |\!) & = & \forall \alpha [\langle\!\langle a : \alpha \rangle\!\rangle]. \alpha
\end{array}
$$

$(\!| a |\!)$ is a *principal constrained type scheme* for a: its intended interpretation is the set of all ground types that a admits.

Properties of constraint generation

Lemma
$\exists\alpha.(\langle\!\langle a : \alpha \rangle\!\rangle \wedge \alpha = \tau) \quad \equiv \quad \langle\!\langle a : \tau \rangle\!\rangle \qquad$ *if* $\alpha \# \tau$.

Lemma
$(\!|a|\!) \leq \tau \quad \equiv \quad \langle\!\langle a : \tau \rangle\!\rangle$.

Lemma
$[x \mapsto (\!|a_1|\!)]\langle\!\langle a_2 : \tau \rangle\!\rangle \quad \equiv \quad \langle\!\langle [x \mapsto a_1]a_2 : \tau \rangle\!\rangle$.

Lemma
$\langle\!\langle \textit{let } x = a_1 \textit{ in } a_2 : \tau \rangle\!\rangle \quad \equiv \quad \langle\!\langle a_1; [x \mapsto a_1]a_2 : \tau \rangle\!\rangle$.

The constraint associated with a let construct is *equivalent* to the constraint associated with its let-normal form.

Complexity

Lemma

The size of $\langle\!\langle a : \tau \rangle\!\rangle$ *is linear in the sum of the sizes of* a *and* τ*.*

Constraint generation can be implemented in linear time and space.

Soundness and completeness

The statement keeps its previous form, back but Γ now contains Damas-Milner type schemes. Since Γ binds variables to type schemes, we define $\phi(\Gamma)$ as the point-wise mapping of $(^{\phi}_{\varnothing})$ to Γ.

Theorem (Soundness and completeness)

Let $\mathrm{fv}(a) = \mathrm{dom}(\Gamma)$*. Then,* $\phi\Gamma \vdash a : \phi\tau$ *if and only if* $\phi, \phi\Gamma \vdash \langle\!\langle a : \tau \rangle\!\rangle$*.*

Summary

Note that

- constraint generation has *linear complexity;*
- constraint generation and constraint solving are *separate;*
- the constraint language remains *small* as the programming language grows.

This makes constraints suitable for use in an efficient and modular implementation.

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

An initial environment

Let Γ_0 stand for *assoc* : $\forall\alpha\beta.\alpha \to \mathit{list}\,(\alpha \times \beta) \to \beta$.

We take Γ_0 to be the *initial environment,* so that the constraints considered next are implicitly wrapped within the context *def* Γ_0 *in* $[]$.

A code fragment

Let a stand for the term

$$\begin{array}{l} \lambda x.\lambda l_1.\lambda l_2. \\ \quad \textit{let assocx} = \textit{assoc}\; x \textit{ in} \\ \quad (\textit{assocx}\;\; l_1, \textit{assocx}\;\; l_2) \end{array}$$

One anticipates that *assocx* receives a polymorphic type scheme, which is instantiated twice at different types...

The generated constraint

Let Γ stand for $x : \alpha_0; l_1 : \alpha_1; l_2 : \alpha_2$. Then, the constraint $\langle\!\langle a : \alpha \rangle\!\rangle$ is (with a few minor simplifications):

$$\exists \alpha_0 \alpha_1 \alpha_2 \beta. \left(\begin{array}{l} \alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta \\ \textit{def } \Gamma \textit{ in} \\ \qquad \textit{let assocx} : \forall \gamma_1 \left[\exists \gamma_2 \left(\begin{array}{l} \textit{assoc} \preceq \gamma_2 \rightarrow \gamma_1 \\ x \preceq \gamma_2 \end{array} \right) \right] . \gamma_1 \textit{ in} \\ \qquad \exists \beta_1 \beta_2 . \left(\begin{array}{l} \beta = \beta_1 \times \beta_2 \\ \forall i \in \{1, 2\}, \quad \exists \gamma_2 . (\textit{assocx} \preceq \gamma_2 \rightarrow \beta_i \wedge l_i \preceq \gamma_2) \end{array} \right) \end{array} \right)$$

Simplification

Constraint solving can be viewed as a *rewriting process* that exploits *equivalence laws*. Because equivalence is, by construction, a *congruence*, rewriting is permitted within an arbitrary context.

For instance, environment access is allowed by the law

$$\textit{let}\ x : \sigma\ \textit{in}\ \mathcal{R}[x \leq \tau] \quad \equiv \quad \textit{let}\ x : \sigma\ \textit{in}\ \mathcal{R}[\sigma \leq \tau]$$

where $\mathcal{R}$ is a context that does not bind x.

Simplification, continued

Thus, within the context *def* $\Gamma_0; \Gamma$ *in* $[]$, the constraint:

$$\left(\begin{array}{c} \mathit{assoc} \leq \gamma_2 \to \gamma_1 \\ x \leq \gamma_2 \end{array} \right)$$

is equivalent to:

$$\left(\begin{array}{c} \exists \alpha\beta.(\alpha \to \mathit{list}\,(\alpha \times \beta) \to \beta = \gamma_2 \to \gamma_1) \\ \alpha_0 = \gamma_2 \end{array} \right)$$

Simplification, continued

By first-order unification, the constraint:

$$\exists \gamma_2. (\exists \alpha\beta. (\alpha \to \textit{list}\,(\alpha \times \beta) \to \beta = \gamma_2 \to \gamma_1) \wedge \alpha_0 = \gamma_2)$$

simplifies down successively to:

$$\exists \gamma_2. (\exists \alpha\beta. (\alpha = \gamma_2 \wedge \textit{list}\,(\alpha \times \beta) \to \beta = \gamma_1) \wedge \alpha_0 = \gamma_2)$$

$$\exists \gamma_2. (\exists \beta. (\textit{list}\,(\gamma_2 \times \beta) \to \beta = \gamma_1) \wedge \alpha_0 = \gamma_2)$$

$$\exists \beta. (\textit{list}\,(\alpha_0 \times \beta) \to \beta = \gamma_1)$$

Simplification, continued

The constrained type scheme:

$$\forall \gamma_1 [\exists \gamma_2.(\mathit{assoc} \le \gamma_2 \to \gamma_1 \wedge x \le \gamma_2)].\, \gamma_1$$

is thus equivalent to:

$$\forall \gamma_1 [\exists \beta.\, (\mathit{list}\,(\alpha_0 \times \beta) \to \beta = \gamma_1)].\, \gamma_1$$

which can also be written:

$$\forall \gamma_1 \beta [\mathit{list}\,(\alpha_0 \times \beta) \to \beta = \gamma_1].\, \gamma_1$$

$$\forall \beta.\mathit{list}\,(\alpha_0 \times \beta) \to \beta$$

Simplification, continued

The initial constraint has now been simplified down to:

$$\exists \alpha_0 \alpha_1 \alpha_2 \beta. \left(\begin{array}{l} \alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta \\ \textit{def } \Gamma \textit{ in} \\ \qquad \textit{let assocx} : \forall \beta.\, \textit{list}\,(\alpha_0 \times \beta) \rightarrow \beta \textit{ in} \\ \qquad \exists \beta_1 \beta_2. \left(\begin{array}{ll} \beta = \beta_1 \times \beta_2 & \\ \forall i & \exists \gamma_2.(\textit{assocx} \preceq \gamma_2 \rightarrow \beta_i \wedge l_i \preceq \gamma_2) \end{array} \right) \end{array} \right)$$

The simplification work spent on *assocx*'s type scheme was well worth the trouble, because we are now going to *duplicate* the simplified type scheme.

Simplification, continued

The sub-constraint:

$$\exists\gamma_2.(\mathit{assocx} \le \gamma_2 \to \beta_i \wedge l_i \le \gamma_2)$$

where $i \in \{1, 2\}$, is rewritten:

$$\begin{aligned}
&\exists\gamma_2.(\exists\beta.(\mathit{list}\,(\alpha_0 \times \beta) \to \beta = \gamma_2 \to \beta_i) \wedge \alpha_i = \gamma_2) \\
&\exists\beta.(\mathit{list}\,(\alpha_0 \times \beta) \to \beta = \alpha_i \to \beta_i) \\
&\exists\beta.(\mathit{list}\,(\alpha_0 \times \beta) = \alpha_i \wedge \beta = \beta_i) \\
&\mathit{list}\,(\alpha_0 \times \beta_i) = \alpha_i
\end{aligned}$$

Simplification, continued

The initial constraint has now been simplified down to:

$$\exists \alpha_0 \alpha_1 \alpha_2 \beta. \left(\begin{array}{l} \alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \textit{def}\ \Gamma\ \textit{in} \\ \qquad \textit{let assocx} : \forall \beta.\, \textit{list}\,(\alpha_0 \times \beta) \to \beta\ \textit{in} \\ \qquad \exists \beta_1 \beta_2. \left(\begin{array}{l} \beta = \beta_1 \times \beta_2 \\ \forall i \quad \textit{list}\,(\alpha_0 \times \beta_i) = \alpha_i \end{array} \right) \end{array} \right)$$

Now, the context *def* Γ *in let assocx* $: \ldots$ *in* $[]$ can be dropped, because the constraint that it applies to contains no occurrences of x, l_1, l_2, or *assocx*.

Simplification, continued

The constraint becomes:

$$\exists \alpha_0\alpha_1\alpha_2\beta.\left(\begin{array}{l} \alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \exists \beta_1\beta_2.\left(\begin{array}{l} \beta = \beta_1 \times \beta_2 \\ \forall i \quad \mathit{list}\,(\alpha_0 \times \beta_i) = \alpha_i \end{array}\right) \end{array}\right)$$

that is:

$$\exists \alpha_0\alpha_1\alpha_2\beta\beta_1\beta_2.\left(\begin{array}{l} \alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \beta = \beta_1 \times \beta_2 \\ \forall i \quad \mathit{list}\,(\alpha_0 \times \beta_i) = \alpha_i \end{array}\right)$$

and, by eliminating a few auxiliary variables:

$$\exists \alpha_0\beta_1\beta_2.(\alpha = \alpha_0 \to \mathit{list}\,(\alpha_0 \times \beta_1) \to \mathit{list}\,(\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2)$$

Simplification, the end

We have shown the following equivalence between constraints:

$$\begin{array}{ll} & \mathit{def}\ \Gamma_0\ \mathit{in}\ \langle\!\langle a : \alpha \rangle\!\rangle \\ \equiv & \exists \alpha_0 \beta_1 \beta_2 . (\alpha = \alpha_0 \rightarrow \mathit{list}\,(\alpha_0 \times \beta_1) \rightarrow \mathit{list}\,(\alpha_0 \times \beta_2) \rightarrow \beta_1 \times \beta_2) \end{array}$$

That is, the *principal type scheme* of a relative to Γ_0 is

$$\begin{array}{lll} (\!|a|\!)_{\Gamma_0} & = & \forall \alpha [\mathit{def}\ \Gamma_0\ \mathit{in}\ \langle\!\langle a : \alpha \rangle\!\rangle].\, \alpha \\ & = & \forall \alpha_0 \beta_1 \beta_2 .\ \ \alpha_0 \rightarrow \mathit{list}\,(\alpha_0 \times \beta_1) \rightarrow \mathit{list}\,(\alpha_0 \times \beta_2) \rightarrow \beta_1 \times \beta_2 \end{array}$$

Rewriting strategies

Again, constraint solving can be explained in terms of a *small-step rewrite system*.

Again, one checks that every step is meaning-preserving, that the system is normalizing, and that every normal form is either literally "*false*" or satisfiable.

Different constraint solving *strategies* lead to different behaviors in terms of complexity, error explanation, etc.

See ATTAPL for details on constraint solving [Pottier and Rémy, 2005]. See Jones [1999] for a different presentation of type inference, in the context of Haskell.

Rewriting strategies

In all reasonable strategies, the left-hand side of a let constraint is simplified *before* the let form is expanded away.

This corresponds, in Algorithm $\mathcal{J}$, to computing a principal type scheme before examining the right-hand side of a let construct.

Complexity

Type inference for ML is DEXPTIME-complete [Kfoury et al., 1990; Mairson, 1990], so any constraint solver has exponential complexity.

Nevertheless, under the hypotheses that *types have bounded size* and let forms have bounded left-nesting depth, constraints can be solved in linear time [McAllester, 2003].

This explains why ML type inference *works well in practice.*

An alternative presentation of constraint generation

Using principal contrained type schemes and the following equivalence

$$(\!|a|\!) \triangleq \forall\alpha\big[\langle\!\langle a : \alpha\rangle\!\rangle\big].\,\alpha \qquad\qquad \langle\!\langle a : \tau\rangle\!\rangle \equiv (\!|a|\!) \le \tau$$

we can also present contraint generation as follows:

$$\begin{array}{rcl}
(\!|x|\!) & = & \forall\alpha[x \le \alpha].\,\alpha \\
(\!|\lambda x.\,a|\!) & = & \forall\alpha_1\alpha_2\big[\textit{def}\ x : \alpha_1\ \textit{in}\ (\!|a|\!) \le \alpha_2\big].\,\alpha_1 \to \alpha_2 \\
& & \text{if } \alpha_1, \alpha_2 \# a \\
(\!|a_1\ a_2|\!) & = & \forall\alpha_1\alpha_2\big[(\!|a_1|\!) \le \alpha_1 \to \alpha_2 \wedge (\!|a_2|\!) \le \alpha_1\big].\,\alpha_2 \\
& & \text{if } \alpha_1, \alpha_2 \# a_1, a_2 \\
(\!|\textit{let}\ x = a_1\ \textit{in}\ a_2|\!) & = & \forall\alpha\big[\textit{let}\ x : (\!|a_1|\!)\ \textit{in}\ (\!|a_2|\!) \le \alpha\big].\,\alpha
\end{array}$$

Type reconstruction

Type inference should not just infer a principal type for an expression. It should also elaborate the implicitly-typed input term into an explicitly-typed one.

Notice that the elaborated term is not unique:

- redundant type abstractions and type applications may be used.
- some non principal type schemes may sometimes be used for local let-bindings.

However, we may seek for a principal derivation in canonical form (as defined in the previous chapter, Damas and Milner's type system).

Type reconstruction

Idea

To perform type reconstruction, it suffices to know the types of let bindings and of function parameters.

In constraints, it suffices to remember def and let-constraints and instantiation constraints $x \preceq \tau$: we may just not remove then during constraint resolution.

We also request that let-constraints be not extruded, so that the binding structure of let-constraints and the scopes of program variables remain as in the original constraint.

Type reconstruction

Preserving the orginal

We modify equivalences used during constraint resolution, so as to preserve the original constraint—and mark it as resolved (in green)

For instance, environment access becomes:

$$\mathit{def}\ x : \sigma\ \mathit{in}\ \mathcal{R}[x \leq \tau] \quad \equiv \quad \mathit{def}\ x : \sigma\ \mathit{in}\ \mathcal{R}[x \leq \tau \wedge \sigma \leq \tau]$$

A binding constraint $\mathit{def}\ x : \sigma\ \mathit{in}\ C$ can be flagged as presolved when x does not appear free in C, except in its resolved subconstraints C:

$$\mathit{def}\ x : \sigma\ \mathit{in}\ C \quad \equiv \quad \mathit{def}\ x : \sigma\ \mathit{in}\ C \qquad\qquad x \# (C \setminus C)$$

A resolved form of a constraint C is an equivalent constraint with the same structure as C that is in solved form after dropping all resolved subconstraints.

Example

Let us reuse a defined above as

$$\lambda x.\, \lambda l_1.\, \lambda l_2.\, \textit{let assocx} = \textit{assoc}\ \ x\ \textit{in}\ (\textit{assocx}\ \ l_1, \textit{assocx}\ \ l_2)$$

The principal type scheme $(\!|a|\!)$ is:

$$\forall \alpha \left[\begin{array}{l} \exists \alpha_0 \alpha_1 \alpha_2 \beta. \\ \quad \left(\begin{array}{l} \alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \textit{def}\ \Gamma\ \textit{in} \\ \quad \textit{let assocx} : \forall \gamma_1 \left[\exists \gamma_2 . \textit{assoc} \le \gamma_2 \to \gamma_1 \wedge x \le \gamma_2 \right] . \gamma_1\ \textit{in} \\ \quad \exists \beta_1 \beta_2 . \left(\begin{array}{l} \beta = \beta_1 \times \beta_2 \\ \forall i \in \{1, 2\}, \ \ \exists \gamma_2 . (\textit{assocx} \le \gamma_2 \to \beta_i \wedge l_i \le \gamma_2) \end{array} \right) \end{array} \right) \end{array} \right] . \alpha$$

where:

- Γ stands for $x : \alpha_0; l_1 : \alpha_1; l_2 : \alpha_2$, and the initial environment
- Γ_0 stands for $\textit{assoc} : \forall \alpha\beta.\, \alpha \to \textit{list}\ (\alpha \times \beta) \to \beta$.

Example

The inner *assocx* type scheme in context Γ can be simplified as follows:

$$
\begin{array}{rl}
& \forall \gamma_1 \left[\exists \gamma_2.\left(\mathit{assoc} \le \gamma_2 \to \gamma_1 \wedge x \le \gamma_2 \right)\right].\gamma_1 \\
\equiv & \forall \gamma_1 \left[\exists \gamma_2.\left(\mathit{assoc} \le \gamma_2 \to \gamma_1 \wedge x \le \gamma_2 \wedge \alpha_0 \le \gamma_2 \right)\right].\gamma_1 \\
\equiv & \forall \gamma_1 \left[\mathit{assoc} \le \alpha_0 \to \gamma_1 \wedge x \le \alpha_0 \right].\gamma_1 \\
\equiv & \forall \gamma_1 \left[\begin{array}{l} \mathit{assoc} \le \alpha_0 \to \gamma_1 \wedge x \le \alpha_0 \\ \forall \alpha\beta.\alpha \to \mathit{list}\,(\alpha \times \beta) \to \beta \le \alpha_0 \to \gamma_1 \end{array} \right].\gamma_1 \\
\equiv & \forall \gamma_1 \left[\begin{array}{l} \mathit{assoc} \le \alpha_0 \to \gamma_1 \wedge x \le \alpha_0 \\ \exists \alpha\beta.(\alpha = \alpha_0 \wedge \mathit{list}\,(\alpha \times \beta) \to \beta = \gamma_1) \end{array} \right].\gamma_1 \\
\equiv & \forall \beta \left[\mathit{assoc} \le \alpha_0 \to \mathit{list}\,(\alpha_0 \times \beta) \to \beta \wedge x \le \alpha_0 \right].\mathit{list}\,(\alpha_0 \times \beta) \to \beta
\end{array}
$$

Example

Simplifying, the remaining instantiations similarly in $(\!|a|\!)$ is equivalent to:

$$\forall \alpha_0 \beta_1 \beta_2 \left[\begin{array}{l} \textit{def}\ \ \Gamma\ \textit{in} \\ \quad \textit{let assocx} : \forall \gamma \left[\begin{array}{l} \textit{assoc} \le \alpha_0 \to \textit{list}\,(\alpha_0 \times \gamma) \to \gamma) \\ x \le \alpha_0 \end{array} \right]. \\ \qquad\qquad\qquad\qquad \textit{list}\,(\alpha_0 \times \gamma) \to \gamma \qquad\qquad\qquad\qquad \textit{in} \\ \quad \forall i \in \{1,2\}, \left(\begin{array}{l} (\textit{assocx} \le \textit{list}\,(\alpha_0 \times \beta_i) \to \beta_i) \\ l_i \le \textit{list}\,(\alpha_0 \times \beta_i))) \end{array} \right) \end{array} \right].$$

$$\alpha_0 \to \textit{list}\,(\alpha_0 \times \beta_1) \to \textit{list}\,(\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2$$

From which, we may read the elaboration of M:

$$\begin{array}{l} \Lambda \alpha_0 \beta_1 \beta_2 .\, \lambda x : \alpha_0 .\, \lambda l_1 : \textit{list}\,(\alpha_0 \times \beta_1).\, \lambda l_2 : \textit{list}\,(\alpha_0 \times \beta_2). \\ \quad \textit{let assocx} = \Lambda \gamma .\, \textit{assoc}\ \alpha_0\ \gamma\ \ x\ \textit{in} \\ \quad (\textit{assocx}\ \beta_1\ \ l_1, \textit{assocx}\ \beta_2\ \ l_2) \end{array}$$

Type abstrations can be read from the principal type scheme.
Type applications can be *locally* inferred from type instantiations.

Type reconstruction, a modular approach

As presented, our type reconstruction is not modular: it builds a *program* typing constraint that it solves and then performs the elaboration from the solved program typing constraint.

Constraint generation is defined independently for each program construct, what about type reconstruction?

Type reconstruction can also be defined this way, for each construct of the language independently, by abstracting over the elaboration of the subconstructs and the solved contrained for the current construct.

See [Pottier, 2014] for details.

This allows to define the constrain solver with elaboration as a library, add new programming constructs without changing the constraint language, or use it for an another language.

Contents

On type annotations

Damas and Milner's type system has *principal types:* at least in the core language, no type information is required.

This is very lightweight, but a bit extreme: sometimes, it is useful to write types down, and use them as *machine-checked documentation.*

Syntax for type annotations

Let us, then, allow programmers to *annotate* a term with a type:

$$a ::= \ldots \mid (a : \tau)$$

Typing and constraint generation are obvious:

$$\textsc{Annot}\ \frac{\Gamma \vdash a : \tau}{\Gamma \vdash (a : \tau) : \tau} \qquad \langle\!\langle (a : \tau) : \tau' \rangle\!\rangle = \langle\!\langle a : \tau \rangle\!\rangle \wedge \tau = \tau'$$

Type annotations are *erased* prior to runtime, so the operational semantics is not affected.

(Erasure of type annotations preserves well-typedness.)

Type annotations are restrictive

The constraint $\langle\!\langle (a : \tau) : \tau' \rangle\!\rangle$ *implies* the constraint $\langle\!\langle a : \tau' \rangle\!\rangle$.

That is, in terms of type inference, *type annotations are restrictive:* they lead to a principal type that is less general, and possibly even to ill-typedness.

For instance, $\lambda x.\, x$ has principal type scheme $\forall \alpha.\, \alpha \to \alpha$, whereas $(\lambda x.\, x : \mathit{int} \to \mathit{int})$ has principal type scheme $\mathit{int} \to \mathit{int}$, and $(\lambda x.\, x : \mathit{int} \to \mathit{bool})$ is ill-typed.

Where

Does it make sense for a type annotation to contain a type variable, as in, say:

$$(\lambda x.\, x : \alpha \to \alpha)$$
$$(\lambda x.\, x + 1 : \alpha \to \alpha)$$
$$\mathit{let}\ f = (\lambda x.\, x : \alpha \to \alpha)\ \mathit{in}\ (f\ 0, f\ \mathit{true})$$

If so, what does it mean?

Short answer: it does not mean anything, because α is unbound. *"There is no such thing as a free variable"* (Alan Perlis).

A longer answer:

It is necessary to specify *how* and *where* type variables are bound.

How is α bound?

If α is *existentially* bound, or *flexible,* then both $(\lambda x.\, x : \alpha \to \alpha)$ and $(\lambda x.\, x + 1 : \alpha \to \alpha)$ should be well-typed.

If it is *universally* bound, or *rigid,* only the former should be well-typed.

Binding type variables

Let's allow programmers to *explicitly bind* type variables:

$$a ::= \ldots \mid \exists \bar{\alpha}.a \mid \forall \bar{\alpha}.a$$

It now makes sense for a type annotation $(a : \tau)$ to contain free type variables.

Terms a can now contain free type variables, so some side conditions have to be updated (e.g., $\bar{\alpha} \# \Gamma, a$ in $\textsc{Gen}$).

Binding type variables

The typing rules (in the implicitly-typed presentation) are as follows:

$$\textsc{Exists}\ \frac{\Gamma \vdash [\vec{\alpha} \mapsto \vec{\tau}]a : \tau}{\Gamma \vdash \exists \bar{\alpha}.a : \tau} \qquad \textsc{Forall}\ \frac{\Gamma \vdash a : \tau \qquad \bar{\alpha} \# \Gamma}{\Gamma \vdash \forall \bar{\alpha}.a : \forall \bar{\alpha}.\tau}$$

$$\left(\textsc{Gen}\ \frac{\Gamma \vdash a : \tau \qquad \bar{\alpha} \# \Gamma, a}{\Gamma \vdash a : \forall \bar{\alpha}.\tau} \right)$$

These constructs are erased prior to runtime.

Why are these rules sound?

Define the erasure of a term, and prove that the erasure of a well-typed term is well-typed:

Rule Exists disappears; Rule Forall becomes rule Gen.

Constraint generation: existential case

Constraint generation for the existential form is straightforward:

$$\langle\!\langle (\exists\bar{\alpha}.a) : \tau \rangle\!\rangle \quad = \quad \exists\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \qquad \text{if } \bar{\alpha} \# \tau$$

The type annotations inside a contain free occurrences of $\bar{\alpha}$. Thus, the constraint $\langle\!\langle a : \tau \rangle\!\rangle$ contains such occurrences as well. They are bound by the existential quantifier.

Constraint generation: existential case

For instance, the expression:

$$\lambda x_1.\, \lambda x_2.\, \exists\alpha.((x_1 : \alpha), (x_2 : \alpha))$$

has principal type scheme $\forall\alpha.\, \alpha \to \alpha \to \alpha \times \alpha$. Indeed, the generated constraint contains the pattern:

$$\exists\alpha.(\langle\!\langle x_1 : \alpha \rangle\!\rangle \wedge \langle\!\langle x_2 : \alpha \rangle\!\rangle \wedge \ldots)$$

which requires x_1 and x_2 to *share* a common (unspecified) type.

Constraint generation: universal case

A term a has type scheme, say, $\forall\alpha.\,\alpha \rightarrow \alpha$ if and only if a has type $\alpha \rightarrow \alpha$ *for every instance of* α, or, equivalently, for an abstract α.

To express this in terms of constraints, we introduce *universal quantification* in the constraint language:

$$C ::= \ldots \mid \forall\alpha.C$$

Its interpretation is standard.

(To solve these constraints, we will use an extension of the unification algorithm called unification under a mixed prefix—see forward.)

The need for universal quantification in constraints arises when polymorphism is *required* by the programmer, as opposed to *inferred* by the system.

Constraint generation: universal case

Constraint generation for the universal form is somewhat subtle. A naive definition *fails* (why?):

$$\langle\!\langle (\forall\bar{\alpha}.a) : \tau \rangle\!\rangle = \forall\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \qquad \text{if } \bar{\alpha} \# \tau \qquad \textit{(Wrong)}$$

This requires τ to be simultaneously equal to *all* of the types that a assumes when $\bar{\alpha}$ varies.

For instance, with this incorrect definition, one would have:

$$\begin{array}{rcl} \langle\!\langle \forall\alpha.(\lambda x.x : \alpha \to \alpha) : \mathit{int} \to \mathit{int} \rangle\!\rangle & = & \forall\alpha.\langle\!\langle (\lambda x.x : \alpha \to \alpha) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\ & \equiv & \forall\alpha.(\langle\!\langle \lambda x.x : \alpha \to \alpha \rangle\!\rangle \wedge \alpha = \mathit{int}) \\ & \equiv & \forall\alpha.(\mathit{true} \wedge \alpha = \mathit{int}) \\ & \equiv & \mathit{false} \end{array}$$

Constraint generation: universal case

A correct definition is:

$$\langle\!\langle (\forall \bar\alpha.a) : \tau \rangle\!\rangle = \forall \bar\alpha.\exists \gamma.\langle\!\langle a : \gamma \rangle\!\rangle \quad \wedge \quad \exists \bar\alpha.\langle\!\langle a : \tau \rangle\!\rangle$$

This requires

- a to be well-typed *for all* instances of $\bar\alpha$ and
- τ to be a valid type for a under *some* instance of $\bar\alpha$.

A problem with this definition is that the term a is duplicated! This can lead to exponential complexity.

Fortunately, this can be avoided modulo a slight extension of the constraint language [Pottier and Rémy, 2003, p. 112]. *The solution defines:*

$$\langle\!\langle \forall \bar\alpha.a : \tau \rangle\!\rangle = \textit{let}\ x : \forall \vec\alpha, \beta[\langle\!\langle a : \beta \rangle\!\rangle].\beta\ \textit{in}\ x \preceq \tau$$

where the new constraint form satisfies the equivalence:

$$\textit{let}\ x : \forall \vec\alpha, \vec\beta[C_1].\tau\ \textit{in}\ C_2 \quad \equiv \quad \forall \vec\alpha.\exists \vec\beta.C_1 \ \wedge\ \textit{def}\ x : \forall \vec\alpha, \vec\beta[C_1].\tau\ \textit{in}\ C_2$$

Type schemes as annotations

Annotating a term with a *type scheme,* rather than just a type, is now just syntactic sugar:

$$(a : \forall \bar{\alpha}.\tau) \quad \text{stands for} \quad \forall \bar{\alpha}.(a : \tau) \qquad\qquad \text{if } \bar{\alpha} \# a$$

In that particular case, constraint generation is in fact simpler:

$$\langle\!\langle (a : \forall \bar{\alpha}.\tau) : \tau' \rangle\!\rangle \;\equiv\; \forall \bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \wedge (\forall \bar{\alpha}.\tau) \leq \tau'$$

(Exercise: check this equivalence.)

Examples

A correct example:

$$
\begin{array}{rl}
 & \langle\!\langle (\exists\alpha.(\lambda x.\, x + 1 : \alpha \to \alpha)) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\
= & \exists\alpha.\langle\!\langle (\lambda x.\, x + 1 : \alpha \to \alpha) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\
\equiv & \exists\alpha.(\alpha = \mathit{int}) \\
\equiv & \mathit{true}
\end{array}
$$

The system *infers* that α must be *int*. Because α is a local type variable, it does not appear in the final constraint.

Examples

An incorrect example:

$$
\begin{array}{rl}
 & \langle\!\langle (\forall \alpha.(\lambda x.\, x + 1 : \alpha \to \alpha)) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\
\Vdash & \forall \alpha.\exists \gamma.\langle\!\langle (\lambda x.\, x + 1 : \alpha \to \alpha) : \gamma \rangle\!\rangle \\
\equiv & \forall \alpha.\exists \gamma.(\alpha = \mathit{int} \wedge \alpha \to \alpha = \gamma) \\
\equiv & \forall \alpha.\alpha = \mathit{int} \\
\equiv & \mathit{false}
\end{array}
$$

The system *checks* that α is used in an abstract way, which is not the case here, since the code implicitly assumes that α is *int*.

Examples

A correct example:

$$
\begin{array}{rl}
& \langle\!\langle (\forall\alpha.(\lambda x.x : \alpha \to \alpha)) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\
= & \forall\alpha.\exists\gamma.\langle\!\langle (\lambda x.x : \alpha \to \alpha) : \gamma \rangle\!\rangle \wedge \exists\alpha.\langle\!\langle (\lambda x.x : \alpha \to \alpha) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\
\equiv & \forall\alpha.\exists\gamma.\alpha \to \alpha = \gamma \wedge \exists\alpha.\alpha = \mathit{int} \\
\equiv & \mathit{true}
\end{array}
$$

The system *checks* that α is used in an abstract way, which is indeed the case here.

It also checks that, if α is appropriately instantiated, the code admits the expected type $\mathit{int} \to \mathit{int}$.

Examples

An incorrect example:

$$
\begin{array}{rl}
& \langle\!\langle \exists\alpha.(\mathit{let}\ f = (\lambda x.\, x : \alpha \to \alpha)\ \mathit{in}\ (f\ 0, f\ \mathit{true})) : \gamma \rangle\!\rangle \\
\equiv & \exists\alpha.(\mathit{let}\ f : \alpha \to \alpha\ \mathit{in} \\
& \qquad \exists\gamma_1\gamma_2.(f \preceq \mathit{int} \to \gamma_1 \wedge f \preceq \mathit{bool} \to \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma)) \\
\equiv & \exists\alpha\gamma_1\gamma_2.(\alpha \to \alpha = \mathit{int} \to \gamma_1 \wedge \alpha \to \alpha = \mathit{bool} \to \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma) \\
\Vdash & \exists\alpha.(\alpha = \mathit{int} \wedge \alpha = \mathit{bool}) \\
\equiv & \mathit{false}
\end{array}
$$

α is bound *outside* the let construct; f receives the monotype $\alpha \to \alpha$.

Examples

A correct example:

$$
\begin{array}{ll}
 & \langle\!\langle \textit{let } f = \exists\alpha.(\lambda x.\, x : \alpha \to \alpha) \textit{ in } (f\ 0, f\ \textit{true}) : \gamma \rangle\!\rangle \\
\equiv & \textit{let } f : \forall\beta[\exists\alpha.(\alpha \to \alpha = \beta)].\,\beta \textit{ in} \\
 & \quad \exists\gamma_1\gamma_2.(f \le \textit{int} \to \gamma_1 \wedge f \le \textit{bool} \to \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma) \\
\equiv & \textit{let } f : \forall\alpha.\,\alpha \to \alpha \textit{ in} \\
 & \quad \exists\gamma_1\gamma_2.(\ldots) \\
\equiv & \exists\gamma_1\gamma_2.(\textit{int} = \gamma_1 \wedge \textit{bool} = \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma) \\
\equiv & \textit{int} \times \textit{bool} = \gamma
\end{array}
$$

α is bound *within* the let construct; the term $\exists\alpha.(\lambda x.\, x : \alpha \to \alpha)$ has the same principal type scheme as $\lambda x.\, x$, namely $\forall\alpha.\,\alpha \to \alpha$; f receives the type scheme $\forall\alpha.\,\alpha \to \alpha$.

Type annotations in the real world

For historical reasons, in Objective Caml, type variables are not explicitly bound. (Retrospectively, that's *bad!*) They are implicitly *existentially* bound at the nearest enclosing toplevel let construct.

In Standard ML, type variables are implicitly *universally* bound at the nearest enclosing toplevel let construct.

In Glasgow Haskell, type variables are implicitly existentially bound within patterns: *'A pattern type signature brings into scope any type variables free in the signature that are not already in scope'* [Peyton Jones and Shields, 2004].

Constraints help understand these varied design choices uniformly.

Type annotations in the real world

The recent versions of OCaml also have a way to specify universally bound type variables, treating them as abtract types:

```
# let f (type a) = ((fun x -> x) : a -> a);;
val f : 'a -> 'a = <fun>

# let f (type a) = ((fun x -> x + 1) : a -> a);;
                              ^
Error: This expression has type a
but an expression was expected of type int
```

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Monomorphic recursion

Recall the typing rule for recursive functions:

$$\frac{\Gamma, f : \tau \vdash \lambda x.\, a : \tau}{\Gamma \vdash \mu f.\lambda x.a : \tau}\ \textsc{FixAbs}$$

It leads to the following derived typing rule:

$$\frac{\begin{array}{c}\Gamma, f : \tau_1 \vdash \lambda x.\, a_1 : \tau_1 \qquad \bar{\alpha} \# \Gamma, a_1 \\ \Gamma, f : \forall \bar{\alpha}.\, \tau_1 \vdash a_2 : \tau_2\end{array}}{\Gamma \vdash \textit{let rec } f\ x = a_1 \textit{ in } a_2 : \tau_2}\ \textsc{LetRec}$$

Any comments?

Monomorphic recursion

These rules require occurrences of f to have *monomorphic type* within the recursive definition (that is, within $\lambda x.\, a_1$).

This is visible also in terms of type inference. The constraint

$$\langle\!\langle \mathit{let\ rec}\ f\ x = a_1\ \mathit{in}\ a_2 : \tau \rangle\!\rangle$$

is equivalent to

$$\mathit{let}\ f : \forall\alpha\beta[\mathit{let}\ f : \alpha \to \beta; x : \alpha\ \mathit{in}\ \langle\!\langle a_1 : \beta \rangle\!\rangle].\, \alpha \to \beta\ \mathit{in}\ \langle\!\langle a_2 : \tau \rangle\!\rangle$$

Monomorphic recursion

This is problematic in some situations, most particularly when defining functions over *nested algebraic data types* [Bird and Meertens, 1998; Okasaki, 1999].

Polymorphic recursion

This problem is solved by introducing *polymorphic recursion*, that is, by allowing μ-bound variables to receive a polymorphic type scheme:

$$\textsc{FixAbsPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a : \sigma}{\Gamma \vdash \mu f.\lambda x.a : \sigma} \qquad \textsc{LetRecPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a_1 : \sigma \qquad \Gamma, f : \sigma \vdash a_2 : \tau}{\Gamma \vdash \textit{let rec } f\ x = a_1 \textit{ in } a_2 : \tau}$$

This extension of ML is due to Mycroft [1984].

In System F, there is no problem to begin with; no extension is necessary.

Polymorphic recursion

Polymorphic recursion alters, to some extent, Damas and Milner's type system.

Now, not only *let-bound,* but also *μ-bound* variables receive type schemes. The type system is no longer equivalent, up to reduction to let-normal form, to simply-typed λ-calculus.

This has two consequences:

- *monomorphization,* a technique employed in some ML compilers [Tolmach and Oliva, 1998; Cejtin et al., 2007], is no longer possible;
- *type inference* becomes problematic!

Polymorphic recursion

Type inference for ML with polymorphic recursion is undecidable [Henglein, 1993]. It is equivalent to the undecidable problem of *semi-unification.*

Polymorphic recursion

Yet, type inference in the presence of polymorphic recursion can be made simple. (How?)

By relying on a *mandatory type annotation*. The rules become:

FixAbsPoly

$$\frac{\Gamma, f : \sigma \vdash \lambda x.\, a : \sigma}{\Gamma \vdash \mu(f : \sigma).\lambda x.a : \sigma}$$

LetRecPoly

$$\frac{\Gamma, f : \sigma \vdash \lambda x.\, a_1 : \sigma \qquad \Gamma, f : \sigma \vdash a_2 : \tau}{\Gamma \vdash \textit{let rec } (f : \sigma) = \lambda x.\, a_1 \textit{ in } a_2 : \tau}$$

The type scheme σ no longer has to be guessed.

With this feature, contrary to what was said earlier back, *type annotations are not just restrictive:* they are sometimes required for type inference to succeed.

Polymorphic recursion

The constraint generation rule becomes:

$$\langle\!\langle \textit{let rec}\,(f : \sigma) = \lambda x.\, a_1 \;\textit{in}\; a_2 : \tau \rangle\!\rangle \quad = \quad ?\textit{let}\; f : \sigma \;\textit{in}\; (\langle\!\langle \lambda x.\, a_1 : \sigma \rangle\!\rangle \wedge \langle\!\langle a_2 : \tau \rangle\!\rangle)$$

It is clear that f receives type scheme σ both *inside and outside* of the recursive definition.

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Unification under a mixed prefix

Unification under a mixed prefix means unification in the presence of both existential and universal quantifiers.

We extend the basic unification algorithm with support for universal quantification.

The solved forms are unchanged: universal quantifiers are always *eliminated.*

Unification under a mixed prefix

In short, in order to reduce $\forall\bar{\alpha}.C$ to a solved form, where C is itself a solved form:

- if a rigid variable is equated with a constructed type, fail;
  $\forall\alpha.\exists\beta\gamma.(\alpha = \beta \rightarrow \gamma)$ is false;
- if two rigid variables are equated, fail;
  $\forall\alpha\beta.(\alpha = \beta)$ is false;
- if a free variable dominates a rigid variable, fail;
  $\forall\alpha.\exists\beta.(\gamma = \alpha \rightarrow \beta)$ is false;
- otherwise, one can decompose C as $\exists\bar{\beta}.(C_1 \wedge C_2)$, where $\bar{\alpha}\bar{\beta} \# C_1$ and $\exists\bar{\beta}.C_2 \equiv$ *true*; then, $\forall\bar{\alpha}.C$ reduces to just C_1.

  $\forall\alpha.\exists\beta\gamma_1.(\beta = \alpha \rightarrow \gamma_1 \wedge \gamma = \gamma_1 \rightarrow \gamma_1)$ reduces to $\exists\gamma_1.(\gamma = \gamma_1 \rightarrow \gamma_1)$, since $\forall\alpha.\exists\beta.(\beta = \alpha \rightarrow \gamma_1)$ is equivalent to *true*.

See [Pottier and Rémy, 2003, p. 109] for details.

Examples

Objective Caml implements a form of unification under a mixed prefix:

```
bash$ ocaml
# let module M : sig val id : 'a → 'a end
                = struct let id x = x + 1 end
  in M.id;;
Values do not match: val id : int → int
is not included in val id : 'a → 'a
```

This example gives rise to a constraint of the form $\forall\alpha.\alpha = int$.

Examples

Here is another example:

```
bash$ ocaml
# let r = ref (fun x → x) in
  let module M : sig val id : 'a → 'a end
                = struct let id = !r end
  in M.id;;
```

Values do not match: val id : '_a → '_a
is not included in val id : 'a → 'a

This example gives rise to a constraint of the form $\exists\beta.\forall\alpha.(\alpha = \beta)$.

Contents

Recursive types

Product and sum types alone do not allow describing *data structures* of *unbounded size*, such as lists and trees.

Indeed, if the grammar of types is $\tau ::= \mathit{unit} \mid \tau \times \tau \mid \tau + \tau$, then it is clear that every type describes a *finite* set of values.

For every k, the type of lists of length at most k is expressible using this grammar. However, the type of lists of unbounded length is not.

Equi- versus iso-recursive types

The following definition is inherently *recursive:*

> *"A list is either empty or a pair of an element and a list."*

We need something like this:

$$\mathit{list}\,\alpha \quad \diamond \quad \mathit{unit} + \alpha \times \mathit{list}\,\alpha$$

But what does $\diamond$ stand for? Is it *equality,* or some kind of *isomorphism?*

Equi- versus iso-recursive types

There are two standard approaches to recursive types, dubbed the *equi-recursive* and *iso-recursive* approaches.

In the equi-recursive approach, a recursive type is *equal* to its unfolding.

In the iso-recursive approach, a recursive type and its unfolding are related via explicit *coercions*.

Equi-recursive types

In the equi-recursive approach, the usual syntax of types:

$$\tau ::= \alpha \mid \mathsf{F}\ \vec{\tau}$$

is no longer interpreted inductively. Instead, types are the *regular trees* built on top of this signature.

Finite syntax for equi-recursive types

If desired, it is possible to use *finite syntax* for recursive types:

$$\tau ::= \alpha \mid \mu\alpha.(\mathsf{F}\ \vec{\tau})$$

We do not allow the seemingly more general $\mu\alpha.\tau$, because $\mu\alpha.\alpha$ is meaningless, and $\mu\alpha.\beta$ or $\mu\alpha.\mu\beta.\tau$ are useless. If we write $\mu\alpha.\tau$, it should be understood that τ is *contractive,* that is, τ is a type constructor application.

For instance, the type of lists of elements of type α is:

$$\mu\beta.(\mathit{unit} + \alpha \times \beta)$$

Finite syntax for equi-recursive types

Each type in this syntax denotes a unique regular tree, sometimes known as its *infinite unfolding.* Conversely, every regular tree can be expressed in this notation (possibly in more than one way).

If one builds a type-checker on top of this finite syntax, then one must be able to *decide* whether two types are *equal,* that is, have identical infinite unfoldings.

This can be done efficiently, either via the algorithm for comparing two DFAs, or by unification. (The latter approach is simpler, faster, and extends to the type inference problem.)

Finite syntax for equi-recursive types

One can also prove [Brandt and Henglein, 1998] that equality is the least congruence generated by the following two rules:

Fold/Unfold

$$\mu\alpha.\tau = [\alpha \mapsto \mu\alpha.\tau]\tau$$

Uniqueness

$$\frac{\tau_1 = [\alpha \mapsto \tau_1]\tau \qquad \tau_2 = [\alpha \mapsto \tau_2]\tau}{\tau_1 = \tau_2}$$

In both rules, τ must be contractive.

This axiomatization does not directly lead to an efficient algorithm for deciding equality, though.

Type soundness for equi-recursive types

In the presence of equi-recursive types, structural induction on types is no longer permitted, but *we never used it* anyway – in soundness proofs. *(We only need it to prove the termination of reduction.)*

It remains true that $\mathsf{F}\ \vec{\tau}_1 = \mathsf{F}\ \vec{\tau}_2$ implies $\vec{\tau}_1 = \vec{\tau}_2$—this was used in our Subject Reduction proofs.

It remains true that $\mathsf{F}_1\ \vec{\tau}_1 = \mathsf{F}_2\ \vec{\tau}_2$ implies $\mathsf{F}_1 = \mathsf{F}_2$—this was used in our Progress proofs.

So, the reasoning that leads to *type soundness* is unaffected.

(Exercise: prove type soundness for the simply-typed λ-calculus in Coq. Then, change the syntax of types from `Inductive` to `CoInductive`.)

Type inference for equi-recursive types

How is type inference adapted for equi-recursive types?

The *syntax* of constraints is unchanged: they remain systems of equations between finite first-order types, without μ's. Their *interpretation* changes: they are now interpreted in a universe of regular trees.

As a result,

- constraint generation is *unchanged;*
- constraint solving is adapted by *removing the occurs check.*

(Exercise: describe solved forms and show that every solved form is either *false* or satisfiable.)

Type inference for equi-recursive types

Here is a function that measures the length of a list:

$$\begin{array}{l} \mu \mathit{length}.\lambda xs.\mathit{case}\ xs\ \mathit{of} \\ \quad\quad \lambda().\,0 \\ \quad [\!]\ \lambda(x, xs).\,1 + \mathit{length}\ xs \end{array}$$

Type inference gives rise to the *cyclic equation:*

$$\beta = \mathit{unit} + \alpha \times \beta$$

where *length* has type $\beta \to \mathit{int}$.

Type inference for equi-recursive types

That is, *length* has *principal type scheme:*

$$\forall\alpha.\,(\mu\beta.\mathit{unit} + \alpha \times \beta) \to \mathit{int}$$

or, equivalently, principal constrained type scheme:

$$\forall\alpha[\beta = \mathit{unit} + \alpha \times \beta].\,\beta \to \mathit{int}$$

The cyclic equation that characterizes lists was never provided by the programmer, but was inferred.

Type inference for equi-recursive types

Objective Caml implements equi-recursive types upon explicit request:

```
bash$ ocaml -rectypes
# type ('a, 'b) sum = Left of 'a | Right of 'b;;
  type ('a, 'b) sum = Left of 'a | Right of 'b

  # let rec length xs =
      match xs with
      | Left () → 0
      | Right (x, xs) → 1 + length xs ;;
  val length : ((unit, 'b * 'a) sum as 'a) → int = ⟨fun⟩
```

Quiz: why is `-rectypes` only an option?

Drawbacks of equi-recursive types

Equi-recursive types are simple and powerful. In practice, however, they are perhaps *too expressive:*

```
bash$ ocaml -rectypes
# let rec map f = function
   | [] → []
   | x :: xs → map f x :: map f xs;;
val map : 'a → ('b list as 'b) → ('c list as 'c) = <fun>

# map (fun x → x + 1) [ 1; 2 ];;
This expression has type int but is used with type 'a list as 'a

# map () [[];[[]]];;
- : 'a list as 'a = [[]; [[]]]
```

Equi-recursive types allow this nonsensical version of map to be accepted, thus delaying the detection of a programmer error.

Half a pint of equi-recursive types

Quiz: why is this accepted?

```
bash$ ocaml
# let f x = x#hello x;;
val f : (< hello : 'a → 'b; .. > as 'a) → 'b = <fun>
```

Iso-recursive types

In the iso-recursive approach, the user is allowed to introduce new *type constructors* D via (possibly mutually recursive) *declarations:*

$$D\,\vec{\alpha} \approx \tau \qquad \text{(where } \mathrm{ftv}(\tau) \subseteq \bar{\alpha}\text{)}$$

Each such declaration adds a unary constructor fold_D and a unary destructor unfold_D with the following types:

$$\begin{array}{rcl} \mathit{fold}_D & : & \forall \bar{\alpha}.\, \tau \rightarrow D\,\vec{\alpha} \\ \mathit{unfold}_D & : & \forall \bar{\alpha}.\, D\,\vec{\alpha} \rightarrow \tau \end{array}$$

and the reduction rule:

$$\mathit{unfold}_D\;(\mathit{fold}_D\; v) \longrightarrow v$$

Iso-recursive types

Ideally, iso-recursive types should not have any runtime cost.

One solution is to compile constructors and destructors away into a target language with equi-recursive types.

Another solution is to see iso-recursive types as a restriction of equi-recursive types where the source language does not have equi-recursive types but instead two unary destructors $fold_D$ and $unfold_D$ with the semantics of the identity function.

Subject reduction does not hold in the source language, but only in the full language with iso-recursive types. Applications of destructors can also be reduced at compile time.

Note that iso-recursive types are less expressive than equi-recursive types, as there is no counter-part to the UNIQUENESS *typing rule.*

Iso-recursive lists

A parametrized, iso-recursive type of lists is:

$$\mathit{list}\,\alpha \approx \mathit{unit} + \alpha \times \mathit{list}\,\alpha$$

The empty list is:

$$\mathit{fold}_{\mathit{list}}\,(\mathit{inj}_1\,())\,:\,\forall\alpha.\,\mathit{list}\,\alpha$$

A function that measures the length of a list is:

$$\left(\begin{array}{l}\mu \mathit{length}.\lambda xs.\mathit{case}\,(\mathit{unfold}_{\mathit{list}}\,xs)\ \mathit{of} \\ \qquad \lambda().\,0 \\ \quad\ [\!]\ \lambda(x, xs).\,1 + \mathit{length}\,xs\end{array}\right) : \forall\alpha.\,\mathit{list}\,\alpha \to \mathit{int}$$

One *folds upon construction* and *unfolds upon deconstruction.*

Type inference for iso-recursive types

In the iso-recursive approach, *types remain finite.* The type *list* α is just an application of a type constructor to a type variable.

As a result, *type inference is unaffected.* The occurs check remains.

Algebraic data types

Algebraic data types result of the fusion of iso-recursive types with structural, labeled products and sums.

This suppresses the *verbosity* of explicit folds and unfolds as well as the *fragility* and inconvenience of numeric indices – instead, named *record fields* and *data constructors* are used.

For instance,

$$\mathit{fold}_{\mathit{list}}\,(\mathit{inj}_1\,())\quad \text{is replaced with}\quad \mathit{Nil}\,()$$

Algebraic data type declarations

An algebraic data type constructor D is introduced via a *record type* or *variant type* definition:

$$D\,\vec{\alpha} \approx \prod_{\ell \in L} \ell : \tau_\ell \qquad \text{or} \qquad D\,\vec{\alpha} \approx \sum_{\ell \in L} \ell : \tau_\ell$$

L denotes a finite set of record labels or data constructors.

Algebraic data type definitions can be mutually recursive.

Effects of a record type declaration

The record type definition $D\,\vec{\alpha} \approx \prod_{\ell \in L} \ell : \tau_\ell$ introduces syntax for *constructing* and *deconstructing* records:

$$C ::= \ldots \mid \{\ell = \cdot\}_{\ell \in L} \qquad\qquad d ::= \ldots \mid \cdot.\ell$$

With the following types

$$\begin{aligned} \{\ell_1 = \cdot, \ldots, \ell_n\} : &\quad \forall \vec{\alpha}.\, \tau_{\ell_1} \to \ldots \tau_{\ell_n} \to D\,\vec{\alpha} \\ \cdot.\ell : &\quad \forall \vec{\alpha}.\, D\,\vec{\alpha} \to \tau_\ell \end{aligned}$$

Effects of a variant type declaration

The variant type definition $D\,\vec{\alpha} \approx \sum_{\ell \in L} \ell : \tau_\ell$ introduces syntax for *constructing* and *deconstructing* variants:

$$C ::= \ldots \mid \ell \qquad\qquad d ::= \ldots \mid \textit{case} \ \cdot\ \textit{of}\,[\ell : \cdot]_{\ell \in L}$$

With the following types:

$$\begin{aligned} \textit{case} \ \cdot\ \textit{of}\,[\ell_1 : \cdot \, [] \ldots \ell_n : \cdot] : &\quad \forall \vec{\alpha}\beta.\, D\,\vec{\alpha} \to (\tau_{\ell_1} \to \beta) \to \ldots (\tau_{\ell_n} \to \beta) \to \beta \\ \ell : &\quad \forall \vec{\alpha}.\, \tau_\ell \to D\,\vec{\alpha} \end{aligned}$$

An example: lists

Here is an algebraic data type of lists:

$$\mathit{list}\ \alpha \approx \mathit{Nil} : \mathit{unit} + \mathit{Cons} : \alpha \times \mathit{list}\ \alpha$$

This gives rise to:

$$\begin{aligned}
\mathit{case}\ \cdot\ \mathit{of}\,[\,\mathit{Nil} : \cdot\ [\!]\ \ldots\ \mathit{Cons} : \cdot\,] : &\quad \forall \alpha\beta.\, \mathit{list}\ \alpha \to (\mathit{unit} \to \beta) \to \\
&\qquad\qquad ((\alpha \times \mathit{list}\ \alpha) \to \beta) \to \beta \\
\mathit{Nil} : &\quad \forall \alpha.\, \mathit{unit} \to \mathit{list}\ \alpha \\
\mathit{Cons} : &\quad \forall \alpha.\, (\alpha \times \mathit{list}\ \alpha) \to \mathit{list}\ \alpha
\end{aligned}$$

A function that measures the length of a list is:

$$\left(\begin{array}{l}
\mu \mathit{length}.\lambda xs.\mathit{case}\ xs\ \mathit{of} \\
\qquad \mathit{Nil} : \lambda().\,0 \\
\quad [\!]\ \mathit{Cons} : \lambda(x, xs).\,1 + \mathit{length}\ xs
\end{array}\right) : \forall \alpha.\, \mathit{list}\ \alpha \to \mathit{int}$$

A word on mutable fields

In Objective Caml, a record field can be marked *mutable*. This introduces an extra binary destructor for writing this field:

$$(\cdot.\ell \leftarrow \cdot) : \forall \vec{\alpha}.\, D\, \vec{\tau} \rightarrow \tau_\ell \rightarrow \mathit{unit}$$

This also makes record construction a destructor since, when fully applied it is *not a value* but it allocates a piece of store and returns its location.

Thus, due to the value restriction, the type of such expressions cannot be generalized.

Contents

HM(X)

Soundness/completeness of type inference are in fact easier to prove if one adopts a *constraint-based specification* of the type system.

In HM(X), judgments take the form

$$C, \Gamma \vdash a : \tau$$

called a constrained typing judgment and should be read *under the asumption C and typing environment Γ, the program a has type τ.*

Here, C ranges over first-order typing constraints as earlier.

However, we require type constraint to have no free program variables.

In a constrained typing judgment, C constrains free *type* variables of the judgment while Γ provides the types of free *program* variables.

This generalizes Damas and Milner's type system.
See Odersky et al. [1999], Pottier and Rémy [2005], Skalka and Pottier [2002] for a detailed treatment.

HM(X) Entailment and subtyping

Typing rules also use an entailment predicate $C \Vdash C'$ between constraints that is more general than constraint equivalence.

Entailment is defined as expected: $C \Vdash C'$ if and only if any ground assignment that satisfies C also satisfies C'.

Then, two constraints are equivalent iff each one entails the other.

Typing judgments for HM(X) are taken up to constraint equivalence.

The parameter X for HM(X) stands for the logic of the constraints.
We have so far only considered constraints with an equality predicate.
Here, we use a more general subtyping predicate $\leq$ that we assume to be contravariant on arrow types:

$$\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' \quad \equiv \quad \tau_2 \leq \tau_2' \wedge \tau_1' \leq \tau_1$$

HM(X) Typing rules

HM-VAR
$$\frac{\sigma = \Gamma(x) \qquad C \Vdash \exists \sigma}{C, \Gamma \vdash x : \sigma}$$

HM-ABS
$$\frac{C, (\Gamma, x : \tau_0) \vdash a : \tau}{C, \Gamma \vdash \lambda x. a : \tau_0 \to \tau}$$

HM-APP
$$\frac{C, \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad C, \Gamma \vdash a_2 : \tau_2}{C, \Gamma \vdash a_1\ a_2 : \tau_1}$$

HM-LET
$$\frac{\begin{array}{c} C, \Gamma \vdash a_1 : \sigma \\ C, (\Gamma, x : \sigma) \vdash a_2 : \tau \end{array}}{C, \Gamma \vdash \textit{let } x = a_1 \textit{ in } a_2 : \tau}$$

HM-GEN
$$\frac{C \wedge C_0, \Gamma \vdash a : \tau \qquad \vec{\alpha} \# C, \Gamma}{C \wedge \exists \vec{\alpha}. C_0, \Gamma \vdash a : \forall \bar{\alpha}[C_0]. \tau}$$

HM-INST
$$\frac{C, \Gamma \vdash a : \forall \vec{\alpha}[C_0]. \tau}{C \wedge C_0, \Gamma \vdash a : \tau}$$

HM-SUB
$$\frac{C, \Gamma \vdash a : \tau_1 \qquad C \Vdash \tau_1 \leq \tau_2}{C, \Gamma \vdash a : \tau_2}$$

HM-EXISTS
$$\frac{C, \Gamma \vdash a : \tau \qquad \vec{\alpha} \# \Gamma, \tau}{\exists \vec{\alpha}. C, \Gamma \vdash a : \tau}$$

HM(X)

The constraint $\exists \sigma$ when σ is a type scheme $\forall \bar{\alpha}[C_0].\tau$ means $\exists \bar{\alpha}.C_0$, *i.e.* that the type scheme is non empty (in premisse of Rule HM-VAR).

A *valid* judgment is one that has a derivation with those typing rules.

In a valid judgment, C may not be satisfiable.

A program is well-typed in environment Γ if the judgment $C, \Gamma \vdash a : \tau$ is valid and C is *satisfiable*.

HM(=) Compared with ML

When considering equality only constraints, HM(=) is equivalent to ML:

HM(=) is a conservative extension of ML:

If Γ *and* τ *contain only Damas-Milner's type schemes, then*

$$\Gamma \vdash a : \tau \in \mathit{ML} \quad \Longleftrightarrow \quad \mathit{true}, \Gamma \vdash a : \tau \in \mathit{HM(=)}$$

HM(=) does not add expressiveness to ML:

If $C, \Gamma \vdash a : \tau \in \mathit{HM(=)}$ *and* φ *is an idempotent solution of* C*,*
then $\Gamma_\varphi \vdash a : \tau_\varphi \in \mathit{ML}$*.*

where $(\cdot)_\varphi$ translates HM(=) type schemes into ML type schemes,
applying the substitution φ on the fly.

PCB(X)

As for ML, there is a syntax directed presentation of typing rules.

However, we may take advantage of program variables in constraints to go one step further and mix the constraint C (without free program variables) and the typing environment Γ into a single constraint C now allowing free program variables.

Judgments take the form $C \vdash a : \tau$ where C both constrains type variables and assigns constrained type schemes to program variables.

The type system, called PCB(X), is equivalent to HM(X)—see Pottier and Rémy [2005] a detailed presentation.

PCB(X)

$$\textsc{pcb-Var}\quad \frac{C \Vdash x \preceq \tau}{C \vdash x : \tau}$$

$$\textsc{pcb-Abs}\quad \frac{C \vdash a : \tau}{\textit{let } x : \tau_0 \textit{ in } C \vdash \lambda x.\, a : \tau_0 \to \tau}$$

$$\textsc{pcb-App}\quad \frac{C_1 \vdash a_1 : \tau_2 \to \tau_1 \qquad C_2 \vdash a_2 : \tau_2}{C_1 \wedge C_2 \vdash a_1\ a_2 : \tau_1}$$

$$\textsc{pcb-Let}\quad \frac{C_1 \vdash a_1 : \tau_1 \qquad C_2 \vdash a_2 : \tau_2}{\textit{let } x : \forall \mathcal{V}[C_1].\, \tau_1 \textit{ in } C_2 \vdash \textit{let } x = a_1 \textit{ in } a_2 : \tau_2}$$

$$\textsc{pcb-Sub}\quad \frac{C \vdash a : \tau_1}{C \wedge \tau_1 \leq \tau_2 \vdash a : \tau_2}$$

$$\textsc{pcb-Exists}\quad \frac{C \vdash a : \tau \qquad \alpha \# \tau}{\exists \alpha.\, C \vdash a : \tau}$$

Soundness and completeness of PCB($=$)

The type inference algorithm for ML is sound and complete for PCB($=$):

- *Soundness:* $\langle\!\langle a : \tau \rangle\!\rangle \vdash a : \tau$.
  The constraint inferred for a typing validates the typing.
- *Completeness:* If $C \vdash a : \tau$ then $C \Vdash \langle\!\langle a : \tau \rangle\!\rangle$.
  The constraint inferred for a typing is more general than any constraint that validates the typing.

HM($\leq$)

In the presence of subtyping, we must recheck type soundness.

This has been done for HM($\leq$) itself, but ideally, this should be done in a more general setting, such as an explicitly typed version of System F with subtyping constraints.

Contents

Full type inference

Type inference has long been an open problem for System F, until Wells [1999] showed that it is in fact undecidable by showing it is equivalent to the semi-unification problem which was earlier proved undecidable.

Type-checking in explicitly-typed System F is indeed feasible and easy (still, an implementation must be careful with renaming of variables when applying substitutions).

However, we have seen that programming with fully-explicit types is unpractical.

Several solutions for *partial type inference* are used in practice. They may alleviate the need for a lot of redundant type annotations. However, none of them is fully satisfactory.

Type inference and second-order unification

The full type-inference problem is not directly related to second-order unification but rather to semi-unification.

However, it becomes equivalent to second-order unification if the positions of type abstractions and type applications are explicit. That is, if terms are

$$M ::= x \mid \lambda x : ?.\, M \mid M\ M \mid \Lambda ?.\, M \mid M\ ?$$

where the question marks stand for type variables and types to inferred.

Second-order unification is still undecidable. One solution is to use semi-algorithms, which may not terminate on some cases. This works arguably well in some cases Pfenning [1988].

Another approach is to restrict to unification under a mixed-prefix. Here, simplifications remain complete (don't loose solutions), but the answer may be "I don't know."

This approach is often used in interactive theorem provers.

Implicit type arguments

Derived from this solution, one can add decorations to let-bindings to indicate that some type arguments are left implicit.

Then, every occurrence of such a variable automatically adds type applications holes for type parameters at the corresponding positions so that will be inferred using second-order unification, while other type applications remain explicit.

Bidirectional type inference

What makes type-checking easy is that typing rules have an algorithmic reading. This implies that they are syntax directed, but also that judgments can be read as functions where some arguments are inputs and others are output.

Typically, Γ and a would be inputs and τ is an ouput in the judgment $\Gamma \vdash a : \tau$, which we may represent as $\Gamma^{\uparrow} \vdash a^{\uparrow} : \tau^{\downarrow}$.

However, although the rules for simply-typed λ-calculus are syntax directed they do not have an algorithmic reading;

The rule for abstraction is

$$\text{Abs}\;\frac{\Gamma^{\uparrow}, x : \tau_0^{\uparrow} \vdash a : \tau^{\downarrow}}{\Gamma^{\uparrow} \vdash \lambda x.\, a : (\tau_0 \to \tau)^{\downarrow}}$$

Then τ_0 is used both as input in the premise and output in the conclusion.

Bidirectional type inference

However, in some cases, the type of the function may be known, *e.g.* when the function is an argument to an expression of a known type.

In such cases, it suffices to check the proposed type is indeed correct.

Formally, the typing judgment $\Gamma \vdash a : \tau$ may be split into two judgments $\Gamma \vdash a \Downarrow \tau$ to check that a may be assigned the type τ and $\Gamma \vdash a \Uparrow \tau$ to infer the type τ of a.

Bidirectional type inference — simple types

$$\textsc{Var-I}\quad \frac{\tau = \Gamma(x)}{\Gamma \vdash x \Uparrow \tau} \qquad \textsc{Abs-C}\quad \frac{\Gamma, x : \tau_0 \vdash a \Downarrow \tau}{\Gamma \vdash \lambda x.\, a \Downarrow \tau_0 \to \tau} \qquad \textsc{App-I}\quad \frac{\Gamma \vdash a_1 \Uparrow \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 \Downarrow \tau_2}{\Gamma \vdash a_1\, a_2 \Uparrow \tau_1}$$

$$\textsc{I-C}\quad \frac{\Gamma \vdash a \Uparrow \tau}{\Gamma \vdash a \Downarrow \tau} \qquad \textsc{Annot-I}\quad \frac{\Gamma \vdash a \Downarrow \tau}{\Gamma \vdash (a : \tau) \Uparrow \tau} \qquad \textsc{Abs-I}\quad \frac{\Gamma, x : \tau_0 \vdash a \Uparrow \tau}{\Gamma \vdash \lambda x{:}\tau_0.\, a \Uparrow \tau_0 \to \tau}$$

Checking mode can use inference mode.

Annotations turn inference mode into checking mode.

Annotations on type abstractions enable the inference mode.

Bidirectional type inference — Simple types

Example: Let τ be $(\tau_1 \to \tau_1) \to \tau_2$. and Γ be $f : \tau$

$$
\dfrac{
  \dfrac{}{\Gamma \vdash f \Uparrow \tau}\ \textsc{Var-I}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{}{\Gamma, x : \tau_1 \vdash x \Uparrow \tau_1}\ \textsc{Var-I}
    }{\Gamma, x : \tau_1 \vdash x \Downarrow \tau_1}\ \textsc{C-I}
  }{\Gamma \vdash \lambda x.\, x \Downarrow \tau_1 \to \tau_1}\ \textsc{Abs-C}
}{
  \dfrac{\Gamma \vdash f\ (\lambda x.\, x) \Uparrow \tau_2}{\Gamma \vdash f\ (\lambda x.\, x) \Downarrow \tau_2}\ \textsc{I-C}
}\ \textsc{App-I}
$$

$$
\dfrac{\Gamma \vdash f\ (\lambda x.\, x) \Downarrow \tau_2}{\varnothing \vdash \lambda f : \tau.\, f\ (\lambda x.\, x) \Downarrow \tau \to \tau_2}\ \textsc{Abs-C}
$$

Bidirectional type inference

Polymorphic types

The method can be extended to deal with polymorphic types.

The idea is due to [Cardelli, 1993] and is still being improved [Dunfield, 2009]. However, it is quite complicated.

Predicative polymorphism is an interesting subcase where partial type inference can be reduced to typing constraints under a mixed prefix. Unfortunately, predicative polymorphism is too restrictive for programming languages (See [Rémy, 2005]).

A simpler approach proposed by Pierce and Turner [2000] and improved by Odersky et al. [2001] is to perform bidirectional type inference only from a small context surounding each node.

Interestingly, bidrectional type inference can easily be extended to work in the presence of subtyping (by constrast with methods based on second-order unification).

Partial type inference

MLF

MLF follows another approach that amounts to performing first-order unification of higher-order types.

- only parameters of functions that are used polymorphically need to be annotated.
- type abstractions and type annotation are always implicit.

However, MLF goes beyond System F: for the purpose of type inference, it introduces richer types that enable to write "more principal types", but that are also harder to read. See [Rémy and Yakobowski, 2008].

The type inference method for MLF can be seen as a generalization of type constraints for ML to handle polymorphic types—still with first-order unification.

Overloading

Contents

What is overloading?

Overloading occurs when at some program point, several definitions for a same identifier are visible simultaneously.

An interpretation of the program (and a fortiori a run of the program) must choose the definition that applies at this point. This is called *overloading resolution*, which may use very different strategies and techniques.

All sorts of identifiers may be subject to overloading: variables, labels, constructors, types, etc.

Overloading must be distinguished from *shadowing* of identifiers by normal scoping rules, where in this case, a new definition may just shadow an older one and temporarily become the only one visible.

Why use overloading?

Naming convenience

It avoids name mangling, such as suffixing similar names by type information: printing functions, *e.g.* print_int, print_string, *etc.*; numerical operations, *e.g.* $+$, $+_{\bullet}$); or numerical constants *e.g.* 0, $0_{\bullet}$

Modularity

To avoid name clashing, the naming discipline (including name mangling conventions) must be known globally. Isolated identifiers with no particular naming convention may still interfere between different developments and cannot be used together unless fully qualified.

To think more abstractly

In terms of operations rather than of particular implementations.
For instance, calling to_string conversion lets the system check whether one definition is available according to the type of the argument.

Why use overloading?

Type dependent functions

A function defined on $\tau[\alpha]$ for all α may have an implementation depending on the type of α. For instance, a marshalling function of type $\forall \alpha. \alpha \to string$ may execute a different code for each base type α.

Ad hoc polymorphism

Overloaded definitions may be *ad hoc*, *i.e.* completely unrelated for each type, or just share a same type schema.

For instance, 0 could mean either the integer zero or the empty list. The symbol $\times$ could mean either integer product or string concatenation.

Why use overloading?

Type dependent functions

A function defined on $\tau[\alpha]$ for all α may have an implementation depending on the type of α. For instance, a marshalling function of type $\forall\alpha.\,\alpha \to string$ may execute a different code for each base type α.

Polytypic polymorphism

Overloaded definitions depend solely on the *type structure* (on whether it is a sum, a product, *etc.*) and can thus be derived mechanically for all types from their definitions on base types.

Typical examples of polytypic functions are marshalling functions or the generation of random values for arbitrary types, *e.g.* as used in Quickcheck for Haskell.

Different forms of overloading

There are many variants of overloading, which can be classified by how overloading is *introduced* and *resolved*.

What are the restrictions on overloading definitions?

- None, *i.e.* arbitrary definitions can be overloaded!
- Can just functions or any definition be overloaded? *e.g.* can numerical values be overloaded?
- Are all overloaded definitions of the same name instances of a common type scheme? Are these type schemes arbitrary?
- Are overloaded definitions primitive (pre-existing), automatic (generated mechanically from other definitions), or user-defined?
- Can overloaded definitions overlap?
- Can overloaded definitions have a local scope?

How is overloading resolved?

How is overloading resolved defined?

- up to subtyping?
- *static* or *dynamic*?

Static resolution (rather simple)

- Overloaded symbols can/must be statically replaced by their implementations at the appropriate types.
- This does not increase expressiveness, but may still significantly reduce verbosity.

How is overloading resolved?

How is overloading resolution defined?

- up to subtyping?
- *static* or *dynamic*?

Dynamic resolution (more involved)

This is required when the choice of the implementation depends on the dynamic of the program execution. For example, the resolution at a program point in a polymorphic function may depend on the type of its argument so that different calls can make different choices.

The resolution is driven by information made available at runtime:

- it can be full or partial type information, or extra values (tags, dictionaries, *etc.*) correlated to types instead of types themselves.
- it can be attached to normal values or passed as extra arguments.

Static resolution

Examples

In SML

Overloaded definitions are primitive (for numerical operators), and automatic (for record accesses).

Typechecking fails if overloading cannot be resolved at outermost let-definitions. For example, **let** twice $x = x + x$ is rejected in SML, at toplevel, as $+$ could be the addition on either integers or floats.

In Java?

Static resolution

In Java

Overloading is not primitive but automatically generated by subtyping. When a class extends another one and a method is redefined, the older definition is still visible, hence the method is overloaded.

Overloading is resolved at compile time by choosing the most specific definition. There is always a best choice—according to static knowledge.

An argument may have a runtime type that is a subtype of the best known compile-time type, and perhaps a more specific definition could have been used if overloading were resolved dynamically.

This is often a source of confusion for Java programmers.

```
class A               { int bin(A y) { return 1; } }
class B extends A     { int bin(B y) { return 2; } }
A aa = new A(); B bb = new B(); A ab = bb;
```

$x.bin(y)$? when $x, y \in \{aa, bb, ab\}$

Static resolution — Limits

It does not fit well with first-class functions and polymorphism:

For example, $\lambda x. x + x$ is rejected when $+$ is overloaded, as it cannot be statically resolved. The function must be specialized at some type at which $+$ is defined.

This argues in favor of some form of dynamic overloading:
dynamic overloading allows to delay resolution of overloaded symbols until polymorphic functions have been sufficiently specialized.

How is dynamic resolution implemented?

Three main techniques for dynamic resolution

- Pass types at runtime and dispatch on the runtime type, using a general typecase construct.
- Tag values with their types—or, usually, an approximation of their types—and dispatch on these tags.
  (This is one possible approach to object-orientation where objects may be tagged with the class they belong to.)
- Pass the appropriate implementations at runtime as extra arguments, usually grouped in *dictionaries* of implementations.

Dynamic resolution — Type passing semantics

Dispatch on runtime type

- Use an explicitly-typed calculus (*e.g.* System F)
- Add a typecase function.
- The runtime cost of typecase may be high, unless type patterns are significantly restricted.
- By default, one pays even when overloading is not used.
- Monomorphization may be used to reduce type matching statically.
- Ensuring exhaustiveness of type matching is difficult.

ML& (Castagna)

- System F + intersection types + subtyping + type matching
- An expressive type system that keeps track of exhaustiveness; type matching functions are first-class and can be extended or overridden.
- Allows overlapping definitions with a best match resolution strategy.

Dynamic resolution — Type erasing semantics

Passing unresolved implementations as extra arguments

- Abstract over unresolved overloaded symbols and pass them around as extra arguments.
  Hopefully, overloaded symbols can be resolved when their types are sufficiently specialized and before they are actually needed.
  In short, *let* $f = \lambda x.\, x + x$ *in* a can be elaborated into
  let $f = \lambda(+).\, \lambda x.\, x + x$ *in* a. Then, the application of f to a float in a *e.g.* $f\ 1.0$ can be elaborated into $f\ (+.)\ 1.0$.
- This can be done based on the typing derivation.
- After elaboration, types are no longer needed and can be erased.
- Monomorphization or other simplifications may reduce the number of abstractions and applications introduced by overloading resolution.

Dynamic resolution — Type erasing semantics

This has been explored under different facets in the context of ML:

- Type classes, introduced in [1989] by Wadler and Blott are the most popular and widely explored framework of this kind.
- Other contemporary proposals were proposed by Rouaix [1990] and Kaes [1992].
- Tentative simplifications of type classes have been made [Odersky et al., 1995] but did not take over, because of their restrictions.
- Other works have tried to relax some restrictions [Morris and Jones, 2010]

We present Mini-Haskell that contains the essence of Haskell.

Type-classes overloading style can also be largely mimicked with implicit module arguments [White et al., 2014] with a few drawbacks but also many advantages.

Contents

Mini-Haskell

Mini Haskell is a simplification of Haskell to avoid most of the difficulties of type classes while keeping their essence:

- single parameter type classes
- no overlapping instance definitions

It is close to *A second look at overloading* by Odersky et al. in terms of expressiveness and simplicity—but closer to Haskell in style: it can be easily generalized by lifting restrictions without changing the framework.

Our version of Mini-Haskell is explicitly typed. We present:

- Some examples in Mini-Haskell.
- Elaboration of Mini-Haskell into (the ML subset of) System F.
- An implicitly-typed version with type inference.

Mini-Haskell Example Implicitly/Explicitly Typed

$$
\begin{array}{l}
\textbf{class}\ \mathit{Eq}\ X\quad \{\ \mathsf{equal} : X \to X \to \mathsf{Bool}\ \} \\
\textbf{inst}\ \ \mathit{Eq}\ \mathsf{Int}\ \ \{\ \mathsf{equal} = \mathsf{primEqInt}\ \} \\
\textbf{inst}\ \ \mathit{Eq}\ \mathsf{Char}\ \{\ \mathsf{equal} = \mathsf{primEqChar}\ \} \\
\textbf{inst}\ \ \Lambda(X)\ \mathit{Eq}\ X \Rightarrow \mathit{Eq}\ (\mathsf{List}\ (X)) \\
\quad \{\ \mathsf{equal} = \lambda(l_1 : \mathsf{List}\ X)\ \lambda(l_2 : \mathsf{List}\ X)\ \mathsf{match}\ l_1,\ l_2\ \mathsf{with} \\
\qquad\qquad \mid\ [],[]\ \to \mathsf{true}\ \mid\ [],\_\ \mid\ \_,[]\ \to \mathsf{false} \\
\qquad\qquad \mid\ h_1{::}t_1,\ h_2{::}t_2 \to \mathsf{equal}\ X\ h_1\ h_2\ \&\&\ \mathsf{equal}\ (\mathsf{List}\ X)\ t_1\ t_2\ \}
\end{array}
$$

This code:

- declares a class (dictionary) of type Eq(X) that contains definitions for equal : $X \to X \to$ Bool,
- creates two concrete instances (dictionaries) of type *Eq* Int and *Eq* Char,
- may create a concrete instance of type *Eq* (List(X)) for any instance of type *Eq*(X)

Example — Elaboration into explicit dictionaries

$$\textbf{class}\ \mathit{Eq}\ X \quad \{\ \text{equal} : X \to X \to \text{Bool}\ \}$$

$$\begin{array}{l}
\textbf{inst}\ \ \mathit{Eq}\ \text{Int}\ \ \{\ \text{equal} = \text{primEqInt}\ \} \\
\textbf{inst}\ \ \mathit{Eq}\ \text{Char}\ \{\ \text{equal} = \text{primEqChar}\ \} \\
\textbf{inst}\ \ \Lambda(X)\ \mathit{Eq}\ X \Rightarrow \mathit{Eq}\ (\text{List}\ (X)) \\
\quad \{\ \text{equal} = \lambda(l_1 : \text{List}\ X)\ \lambda(l_2 : \text{List}\ X)\ \text{match}\ l_1,\ l_2\ \text{with} \\
\qquad\qquad \mid\ [],[]\ \to \text{true}\ \mid\ [],_\ \mid\ [],_\ \to \text{false} \\
\qquad\qquad \mid\ h_1 :: t_1,\ h_2 :: t_2 \to \text{equal}\ X\ h_1\ h_2\ \&\&\ \text{equal}\ (\text{List}\ X)\ t_1\ t_2\ \}
\end{array}$$

Becomes:

$$\begin{array}{l}
\textbf{type}\ \ \mathit{Eq}\ (X) = \{\ \text{equal} : X \to X \to \text{Bool}\ \} \\
\textbf{let}\ \ \ \text{equal}\ X\ (\mathit{EqX} : \mathit{Eq}\ X) : X \to X \to \text{Bool} = \mathit{EqX}.\text{equal}
\end{array}$$

$$\begin{array}{l}
\textbf{let}\ \ \mathit{EqInt} : \mathit{Eq}\ \text{Int}\ \ = \{\ \text{equal} = \text{primEqInt}\ \} \\
\textbf{let}\ \ \mathit{EqChar} : \mathit{Eq}\ \text{Char} = \{\ \text{equal} = \text{primEqChar}\ \} \\
\textbf{let}\ \ \mathit{EqList}\ X\ (\mathit{EqX} : \mathit{Eq}\ X) : \mathit{Eq}\ (\text{List}\ X) \\
\quad \{\ \text{equal} = \lambda(l_1 : \text{List}\ X)\ \lambda(l_2 : \text{List}\ X)\ \ \text{match}\ l_1,\ l_2\ \text{with} \\
\qquad\qquad \mid\ [],[]\ \to \text{true}\ \mid\ [],_\ \mid\ [],_\ \to \text{false} \\
\qquad\qquad \mid\ h_1 :: t_1,\ h_2 :: t_2 \to \\
\qquad\qquad\quad \text{equal}\ X\ \mathit{EqX}\ h_1\ h_2\ \&\&\ \text{equal}\ (\text{List}\ X)\ (\mathit{EqList}\ X\ \mathit{EqX})\ t_1\ t_2\ \}
\end{array}$$

Example — Class Inheritance

Classes may themselves depend on other classes (called superclasses):

$$\begin{array}{l} \textbf{class } \mathit{Eq}\ X \Rightarrow \mathit{Ord}\ (X)\ \{\ \mathsf{lt} : X \to X \to \mathsf{Bool}\ \} \\ \textbf{inst }\ \ \mathit{Ord}\ \mathsf{Int}\ \{\ \mathsf{lt} = (<)\ \} \end{array}$$

This declares a new class (dictionary) $\mathit{Ord}\ X$ that depends on a dictionary $\mathit{Eq}\ X$ and contains a method $\mathsf{lt} : X \to X \to \mathsf{Bool}$.

The instance definition builds a dictionary $\mathit{Ord}\ \mathsf{Int}$ from the existing dictionary $\mathit{Eq}\ \mathsf{Int}$ and the primitive $(<)$ for lt.

The two declarations are elaborated into:

$$\begin{array}{l} \textbf{type } \mathit{Ord}\ X = \{\ \mathit{Eq} : \mathit{Eq}\ X;\ \mathsf{lt} : X \to X \to \mathsf{Bool}\ \} \\ \textbf{let }\ \ \mathit{EqOrd}\ X\ (\mathit{OrdX} : \mathit{Ord}\ X) : \mathit{Eq}\ X = \mathit{OrdX}.\mathit{Eq} \\ \textbf{let }\ \ \mathsf{lt}\ \ X\ (\mathit{OrdX} : \mathit{Ord}\ X) : X \to X \to \mathsf{Bool} = \mathit{OrdX}.\mathsf{lt} \end{array}$$

$$\textbf{let } \mathit{OrdInt} : \mathit{Ord}\ \mathsf{Int} = \{\ \mathit{Eq} = \mathit{EqInt};\ \mathsf{lt} = (<)\ \}$$

Mini Haskell

Overloading

An overloaded function search is defined as follows:

$$
\begin{array}{l}
\textbf{let rec } \mathsf{leq} : \forall(\mathsf{X})\ \mathsf{Ord}\ \mathsf{X} \Rightarrow \mathsf{X} \to \mathsf{List}\ \mathsf{X} \to \mathsf{Bool} = \\
\quad \Lambda(\mathsf{X})\ \lambda(\mathsf{x} : \mathsf{X})\ \lambda(\mathsf{l} : \mathsf{ListX}) \\
\qquad \mathsf{match}\ \mathsf{l}\ \mathsf{with}\ [] \to \mathsf{true} \\
\quad\ \ \mid \mathsf{h{::}t} \to (\mathsf{lt}\ \ \mathsf{x}\ \mathsf{h} \mathbin{||} \mathsf{equal}\ \mathsf{x}\ \mathsf{h})\ \&\&\ \mathsf{leq}\ \mathsf{x}\ \mathsf{t}
\end{array}
$$

$$\textbf{let } \mathsf{b} = \mathsf{leq}\ \mathsf{Int}\ 1\ [1;\ 2;\ \ 3];;$$

This elaborates into:

$$
\begin{array}{l}
\textbf{let rec } \mathsf{leq}\ \mathsf{X}\ (\mathit{OrdX} : \mathsf{Ord}\ \mathsf{X})\ (\mathsf{x} : \mathsf{X})\ (\mathsf{l} : \mathsf{ListX}) : \mathsf{Bool} = \\
\quad \mathsf{match}\ \mathsf{l}\ \mathsf{with} \mid [] \to \mathsf{true} \\
\quad \mid \mathsf{h{::}t} \to (\mathsf{lt}\ \ \mathsf{X}\ \mathit{OrdX}\ \mathsf{x}\ \mathsf{h} \mathbin{||} \mathsf{equal}\ \mathsf{X}\ (\mathit{EqOrd}\ \mathsf{X}\ \mathit{OrdX})\ \mathsf{x}\ \mathsf{h}) \\
\qquad\qquad\quad \&\&\ \mathsf{leq}\ \mathsf{X}\ \mathit{OrdX}\ \mathsf{x}\ \mathsf{t}
\end{array}
$$

$$\textbf{let } \mathsf{b} = \mathsf{leq}\ \mathsf{Int}\ \mathit{OrdInt}\ 1\ [1;\ 2;\ 3];;$$

That is, the code in green is inferred.

Contents

Mini Haskell

We restrict to single parameter classes.

Class and instance declarations are restricted to the toplevel.
Their scope is the whole program.

Mini Haskell

In practice, a program is composed of *interleaved*

- class declarations,
- instance definitions,
- function definitions,

given *in any order* and

- ending with an expression.

Instance and function definitions are interpreted recursively.
Hence, their definition order does not matter.

For simplification, we assume that instance definitions do not depend on function definitions, which may then come last as part of the expression in a recursive let-binding.

Mini Haskell

In practice, a program is composed of *sequences of*

- class declarations,
- instance definitions,

given *in this order* and

- ending with an expression.

Instance definitions are interpreted recursively; their order does not matter.

We may assume, *w.l.o.g.*, that instance definitions come after all class declarations.

The order of class declaration matters, since they may only refer to other class constructors that have been previously defined.

Mini Haskell

Source programs p are of the form:

$$p ::= H_1 \dots H_p \; h_1 \dots h_q \; M$$

$$H ::= \textsf{\textit{class}}\ \vec{P} \Rightarrow \mathsf{K}\,\alpha\ \{\rho\}$$
$$\rho ::= u_1 : \tau_1, \dots u_m : \tau_m$$

$$h ::= \textsf{\textit{inst}}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\,(\mathsf{G}\,\vec{\beta})\ \{r\}$$
$$r ::= u_1 = M_1, \dots u_k = M_k$$

$$P ::= \mathsf{K}\,\alpha \qquad Q ::= \mathsf{K}\,\tau \qquad \sigma ::= \forall \vec{\alpha}. \vec{Q} \Rightarrow T \qquad T ::= \tau \mid Q$$

Letter u ranges over overloaded symbols.

Class constructors K may appear in Q but not in τ.
Only regular type constructors G may appear in τ.

We write $\forall \vec{\alpha}. Q_1 \Rightarrow \dots Q_m \Rightarrow T$ for $\forall \vec{\alpha}. Q_1, \dots Q_m \Rightarrow T$
and see $\Rightarrow$ as an annotated version of $\rightarrow$.

Mini Haskell

Source programs p are of the form:

$$p ::= H_1 \ldots H_p \; h_1 \ldots h_q \; M$$

$$H ::= \mathit{class}\ \vec{P} \Rightarrow \mathsf{K}\,\alpha\ \{\rho\}$$
$$\rho ::= u_1 : \tau_1, \ldots u_m : \tau_m$$

$$h ::= \mathit{inst}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\,(\mathsf{G}\,\vec{\beta})\ \{r\}$$
$$r ::= u_1 = M_1, \ldots u_k = M_k$$

$$P ::= \mathsf{K}\,\alpha \qquad Q ::= \mathsf{K}\,\tau \qquad \sigma ::= \forall \vec{\alpha}.\vec{Q} \Rightarrow T \qquad T ::= \tau \mid Q$$

The sequence $\vec{P}$ in class and instance definitions is a *typing context*. Each clause $\vec{P}$ is of the form $\mathsf{K}'\ \alpha'$ and can be read as an assumption "*given a dictionary K' of type α'...*"

The restriction to types of the form $\mathsf{K}'\ \alpha'$ in typing contexts and class declarations, and to types of the form $\mathsf{K}\,(\mathsf{G}\,\vec{\beta})$ in instances are for simplicity. Generalizations are discussed later.

Target language

System F, extended with record types, let-bindings, and let-rec.

Records are provided as data types. They are used to represent dictionaries. Record labels represent overloaded symbols u.

We may also use overloaded symbols u as variables.
This amounts to reserving a subset of variables x_u indexed by overloaded symbols, but just writing u as a shortcut for x_u.

We use letter N instead of M for elaborated terms, to distinguish them from source terms.

Class declarations

$$H \triangleq \textit{class}\; K_1\,\alpha, \ldots K_p\,\alpha \Rightarrow K\,\alpha\;\{\rho\}$$

A class declaration H defines a class constructor K.

Every class (constructor) K must be defined by one and only one class declaration. So we may say that H is the declaration of K.

Classes K_i's are superclasses of K and we write $K_i \prec K$.

Class definitions must respect the order $\prec$ *(acyclic)*

The dictionary of K will contain a sub-dictionary for each superclass K_i.

All K_i's are independent in a *typing context*: there does not exists i and j such that $K_j \prec K_i$.

Indeed, if $K_j \prec K_i$, then K_i dictionary would contain a sub-dictionary for K_j, to which K has access via K_i so K does not itself need dictionary K_j.

Class declarations

$$H \quad \triangleq \quad \textsf{\textit{class}}\; K_1\,\alpha, \ldots K_p\,\alpha \Rightarrow \textsf{\textit{K}}\,\alpha\,\{\rho\}$$

The row type ρ is of the form

$$u_1 : \tau_1, \ldots\; u_m : \tau_m$$

and declares *overloaded symbols* u_i (also called *methods*) of class *K*.

An overloaded symbol cannot be declared twice in the same class and must be declared only in one class.

Types τ_i's must be closed with respect to α.

Each class instance will contain a definition for each method.

Class declarations

Elaboration

$$H \triangleq \textit{class}\; K_1\,\alpha, \ldots K_p\,\alpha \Rightarrow K\,\alpha\,\{\rho\}$$

Its elaboration consists of a record type declaration to represent the dictionary and the definition of accessors for each field of the record.

The row ρ only lists methods $u_1 : \tau_1, \ldots u_m : \tau_m$. We extend it with sub-dictionary fields and define ρ^K to be $\rho, u^K_{K_1} : K_1\,\alpha, \ldots u^K_{K_p} : K_p\,\alpha$. Thus ρ^K is of the form $u_1 : T_1, \ldots u_n : T_n$. We introduce:

- a record type definition $K\,\alpha \approx \{u_1 : T_1, \ldots u_n : T_n\}$,
- for each i in $1..n$ we define the accessor to field u_i:
  - let N_i be $\Lambda\alpha.\,\lambda z{:}K\,\alpha.\,(z.u_i)$.
  - let σ_i be $\forall\alpha.\,K\,\alpha \Rightarrow T_i$, *i.e.* the type of N_i
  - let $\mathcal{R}_i$ be the program context $\textit{let}\; u_i : \sigma_i = N_i \;\textit{in}\; [\,]$.

Then, $[\![H]\!]$ is $\mathcal{R}_1 \circ \ldots \mathcal{R}_n$ and we write Γ_H for the typing environment $u_1 : \sigma_1 \ldots u_p : \sigma_p$ in the hole of $[\![H]\!]$.

Class declarations

Elaboration

The elaboration $[\![\vec{H}]\!]$ of the sequence of class definitions $\vec{H}$ is the composition of the elaboration of each.

$$[\![H_1 \ldots H_p]\!] \triangleq [\![H_1]\!] \circ \ldots [\![H_p]\!] \triangleq \mathit{let}\ \vec{u} : \vec{\sigma}_u = \vec{N}_u\ \mathit{in}\ []$$

Record type definitions are collected in the program prelude.

We write $\Gamma_{H_1 \ldots H_p}$ for $\Gamma_{H_1}, \ldots \Gamma_{H_p}$.

Instance definitions

$$h \;\triangleq\; \mathit{inst}\,\forall\vec{\beta}.\,K'_1\,\beta_1,\ldots K'_k\,\beta_k \Rightarrow K\,(G\,\vec{\beta})\;\{r\}$$

It defines an instance of a class K.

The *typing context* $K'_1\,\beta_1,\ldots K'_k\,\beta_k$ describes the dictionaries that must be available on type parameters $\vec{\beta}$ to build the dictionary $K\,(G\,\vec{\beta})$.

This is not related to the superclasses of the class K:

For example, in

$$\textbf{inst}\;\;\Lambda(\mathrm{X})\;\mathit{Eq}\;\mathrm{X} \Rightarrow \mathit{Eq}\;(\mathrm{List}\;(\mathrm{X}))$$

An instance of class *Eq* at type X is needed to build an instance of class *Eq* at type List(X), but *Eq* is not a superclass of itself.

Instance definitions

$$h \ \stackrel{\triangle}{=} \ \mathit{inst}\ \forall \vec{\beta}.\ K'_1\ \beta_1, \ldots K'_k\ \beta_k \Rightarrow \boxed{\mathsf{K}\,(\mathsf{G}\,\vec{\beta})}\ \{r\}$$

The typing context describes dictionaries that cannot yet be built because they depend on some unknown type β in $\vec{\beta}$.

We assume that the typing context is such that:

- each β_i is in $\vec{\beta}$
- β_i and β_j may be equal, except if K_i and K_j are related ($i.e.$ $\mathsf{K}_i \prec \mathsf{K}_j$ or $\mathsf{K}_j \prec \mathsf{K}_i$ or $\mathsf{K}_i = \mathsf{K}_j$)
  The reason is, as for class declarations, that it would be useless to require both dictionaries $\mathsf{K}_i\ \beta$ and $\mathsf{K}_j\ \beta$ when they are equal or one is contained in the other.

Such typing contexts are said to be *canonical*.

Instance declarations Elaboration

$$h \triangleq \textit{inst}\ \forall \vec\beta.\, K'_1\ \beta_1, \ldots K'_k\ \beta_k \Rightarrow K\,(G\,\vec\beta)\ \{r\}$$

This instance definition h is elaborated into a triple (z_h, N^h, σ_h) where z_h is an identifier to refer to the elaborated body N^h of type σ_h.

The type σ_h is $\forall \vec\beta.\, K'_1\ \beta_1 \Rightarrow \ldots K'_k\ \beta_k \Rightarrow K\,(G\,\vec\beta)$

The expression N^h builds a dictionary of type $K\,(G\,\vec\beta)$, given $k \geq 0$ dictionaries of respective types $K'_1\ \beta_1, \ldots K'_k\ \beta_k$:

$$\Lambda\vec\beta.\, \lambda(z_1 : K'_1\ \beta_1).\ \ldots \lambda(z_k : K'_k\ \beta_k).$$
$$\{u_1 = N^h_1, \ldots u_m = N^h_m,\ u^K_{K_1} = q_1, \ldots u^K_{K_p} = q_p\}$$

The types of fields are as prescribed by the class definition K:

- N^h_i is the elaboration of M_i where r is $u_1 = M_1, \ldots u_m = M_m$.
- q_i is a dictionary of type $K_i\ (G\,\vec\beta)$ (the i'th subdictionary of K)

(We write z for a variable x that binds a dictionary.)

Elaboration of whole programs

The elaboration of all class instances $[\![\vec{h}]\!]$ is the program context

$$\textit{let rec}\ (\vec{z}_h : \vec{\sigma}_h)\ = \vec{N}^h\ \textit{in}\ [\,]$$

The elaboration of the whole program $\vec{H}\ \vec{h}\ M$ is

$$[\![\vec{H}\ \vec{h}\ M]\!] \triangleq \textit{let}\ \vec{u} : \vec{\sigma}_u = \vec{N}_u\ \textit{in}\ \textit{let rec}\ (\vec{z}_h : \vec{\sigma}_h)\ = \vec{N}^h\ \textit{in}\ N$$

Hence, the expression N and all expressions N^h are typed (and elaborated) in the environment Γ_0 equal to $\Gamma_{\vec{H}}, \Gamma_{\vec{h}}$ where

- $\Gamma_{\vec{H}}$ declares functions to access components of dictionaries (both sub-dictionaries and definitions of overloaded symbols).
- $\Gamma_{\vec{h}}$ equal to $(\vec{z}_h : \vec{\sigma}_h)$ declares functions to build dictionaries (*i.e.* all class instances).

Elaboration of expressions

The elaboration of expressions is defined by a judgment

$$\Gamma \vdash M \rightsquigarrow N : \sigma$$

where Γ is a System-F typing context, M is the source expression, N is the elaborated expression and σ its type in Γ.

In particular, $\Gamma \vdash M \rightsquigarrow N : \sigma$ implies $\Gamma \vdash N : \sigma$ in F.

We write q for dictionary terms, *i.e.* the following subset of F terms:

$$q ::= u \mid z \mid q\ \tau \mid q\ q$$

(u and z are just particular cases of x)

The elaboration of dictionaries is the judgment $\Gamma \vdash q : \sigma$ which is just typability in System F—but restricted to dictionary expressions.

Elaboration of expressions

$$\textsc{Var}\quad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x \leadsto x : \sigma}$$

$$\textsc{Inst}\quad \frac{\Gamma \vdash M \leadsto N : \forall \alpha.\, \sigma}{\Gamma \vdash M\ \tau \leadsto N\ \tau : [\alpha \mapsto \tau]\sigma}$$

$$\textsc{Gen}\quad \frac{\Gamma, \alpha \vdash M \leadsto N : \sigma}{\Gamma \vdash \Lambda\alpha.\, M \leadsto \Lambda\alpha.\, N : \forall \alpha.\, \sigma}$$

$$\textsc{Let}\quad \frac{\Gamma \vdash M_1 \leadsto N_1 : \sigma \qquad \Gamma, x : \sigma \vdash M_2 \leadsto N_2 : \tau}{\Gamma \vdash \textit{let}\ x : \sigma = M_1\ \textit{in}\ M_2 \leadsto \textit{let}\ x : \sigma = N_1\ \textit{in}\ N_2 : \tau}$$

$$\textsc{App}\quad \frac{\begin{array}{c}\Gamma \vdash M_1 \leadsto N_1 : \tau_2 \to \tau_1 \\ \Gamma \vdash M_2 \leadsto N_2 : \tau_2\end{array}}{\Gamma \vdash M_1\ M_2 \leadsto N_1\ N_2 : \tau_1}$$

$$\textsc{Abs}\quad \frac{\Gamma, x : \tau' \vdash M \leadsto N : \tau}{\Gamma \vdash \lambda x{:}\tau'.\, M \leadsto \lambda x{:}\tau'.\, N : \tau' \to \tau}$$

In rule Let, σ must be canonical, *i.e.* of the form $\forall \vec{\alpha}.\, \vec{P} \Rightarrow T$ where $\vec{P}$ is itself empty or canonical (see the definition and also this restriction).

Rules App and Abs do not apply to overloaded expressions of type σ.

Elaboration of overloaded expressions

The interesting rules are the elaboration of missing abstractions and applications of dictionaries.

$$\textsc{OAbs}\;\frac{\Gamma, x : Q \vdash M \rightsquigarrow N : \sigma \qquad x \# M}{\Gamma \vdash M \rightsquigarrow \lambda x : Q. N : Q \Rightarrow \sigma} \qquad \textsc{OApp}\;\frac{\Gamma \vdash M \rightsquigarrow N : Q \Rightarrow \sigma \qquad \Gamma \vdash q : Q}{\Gamma \vdash M \rightsquigarrow N\, q : \sigma}$$

Rule OAbs pushes dictionary abstractions Q in the context Γ as prescribed by the expected type of the argument x.

These may then be used (in addition to dictionary accessors and instance definitions already in Γ) to elaborate dictionaries as described by the premise $\Gamma \vdash q : Q$ of rule OApp.

Elaboration of overloaded expressions

The interesting rules are the elaboration of missing abstractions and applications of dictionaries.

$$\text{OABS}\quad \frac{\Gamma, x : Q \vdash M \rightsquigarrow N : \sigma \qquad x \# M}{\Gamma \vdash M \rightsquigarrow \lambda x : Q.\, N : Q \Rightarrow \sigma} \qquad \text{OAPP}\quad \frac{\Gamma \vdash M \rightsquigarrow N : Q \Rightarrow \sigma \qquad \Gamma \vdash q : Q}{\Gamma \vdash M \rightsquigarrow N\ q : \sigma}$$

Judgment $\Gamma \vdash q : Q$ is just well-typedness in System F, but restricted to dictionary expressions. There is an algorithmic reading of the rule, described further, where Γ and Q are given and q is inferred.

Elaboration of overloaded expressions

The interesting rules are the elaboration of missing abstractions and applications of dictionaries.

OAbs
$$\frac{\Gamma, \boxed{x : Q} \vdash M \rightsquigarrow N : \sigma \qquad x \# M}{\Gamma \vdash M \rightsquigarrow \lambda \boxed{x} : \boxed{Q}. N : \boxed{Q} \Rightarrow \sigma}$$

OApp
$$\frac{\Gamma \vdash M \rightsquigarrow N : \boxed{Q} \Rightarrow \sigma \qquad \boxed{\Gamma \vdash q : Q}}{\Gamma \vdash M \rightsquigarrow N\ \boxed{q} : \sigma}$$

By construction, elaboration produces well-typed expressions: that is $\Gamma_0 \vdash M \rightsquigarrow N : \tau$ implies that is $\Gamma_0 \vdash N : \tau$.

Resuming the elaboration

An instance declaration h of the form:

$$\mathit{inst}\ \forall \vec{\beta}.\ K'_1\ \beta_1, \ldots K'_k\ \beta_k \Rightarrow K\,(G\,\vec{\tau})\ \{u_1 = M_1, \ldots u_m = M_m\}$$

is translated into

$$\begin{array}{l} \Lambda\vec{\beta}.\,\lambda(z_1 : K'_1\ \beta_1).\ \ldots \lambda(z_k : K'_k\ \beta_k). \\ \qquad \{u_1 = N^h_1, \ldots u_m = N^h_m,\ u^K_{K_1} = q_1, \ldots u^K_{K_p} = q_p\} \end{array}$$

where:

- $u^K_{K_i} : Q_i$ are the superclasses fields, $u_i : \tau_i$ are the method fields
- Γ_h is $\vec{\beta}, K'_1\ \beta_1, \ldots K'_k\ \beta_k$
- $\Gamma_0, \Gamma_h \vdash q_i : Q_i$
- $\Gamma_0, \Gamma_h \vdash M_i \rightsquigarrow N_i : \tau_i$

Finally, given the program p equal to $\vec{H}\ \vec{h}\ M$, we elaborate M as N such that $\Gamma_0 \vdash M \rightsquigarrow N : \forall\bar{\alpha}.\tau$.

Notice that $\forall\bar{\alpha}.\tau$ is an unconstrained type scheme. Why?

Let-monomorphization

Otherwise, N could elaborate into an abstraction over dictionaries, *i.e.* it would be a value and never applied!

Where else should we be careful that the *intended* semantics is preserved?

In a call-by-value setting, we must not elaborate applications into abstractions, since it would delay and perhaps duplicate the order of evaluations.

For that purpose, we must restrict rule $\textsc{Let}$ so that either σ is of the form $\forall\bar{\alpha}.\tau$ or M_1 is a value or a variable.

What about call-by-name? and Haskell?

Let-monomorphization

In call-by-name, an application is not evaluated until it is needed. Hence, adding an abstraction in front of an application should not change the evaluation order $M_1\ M_2$.

We must in fact compare:

$$\begin{array}{ll} \textit{let}\ x_1 = \textit{let}\ x_2 = \lambda y.\, V_1\ V_2\ \textit{in}\ [x_2 \mapsto x_2\ q]M_2\ \textit{in}\ M_1 & \quad \mathbf{(1)} \\ \textit{let}\ x_1 = \lambda y.\, \textit{let}\ x_2 = V_1\ V_2\ \textit{in}\ M_2\ \textit{in}\ [x_1 \mapsto x_1\ q]M_1 & \quad \mathbf{(2)} \end{array}$$

The order of evaluation of $V_1\ V_2$ is preserved.

However, Haskell is call-by-need, and not call-by-name!

Hence, applications are delayed as in call-by-name but their evaluation is shared and only reduced once.

The application $V_1\ V_2$ will be reduced once in (2), but as many types as there are occurrences of x_2 in M_2 in (1).

Let-monomorphization

The final result will still be the same in both cases because Haskell is pure, but the intended semantics is changed regarding the efficiency.

Hence, Haskell may also use monomorphization in this case. This is a delicate design choice

(Of course, monomorphization reduces polymorphism, hence the set of typable programs.)

Resuming the elaboration

Sources of failures

The elaboration may fail for several reasons:

- The input expression does not obey one of the restrictions we have requested.
- A typing error may occur during elaboration of an expression.
- Some required dictionary cannot be built.

If elaboration fails, the program p is rejected, indeed.

When elaboration succeeds

When the elaboration of p succeeds it returns $[\![p]\!]$, well-typed in F.

Then, the semantics of p is given by that of $[\![p]\!]$.

Hum... Although terms are explicitly-typed, their elaboration may not be unique! Indeed, there might be several ways to build dictionaries of some given type (see below for details).

In the worst case, a source program may elaborate to completely unrelated programs. In the best case, all possible elaborations are *equivalent* programs and we say that the elaboration is *coherent*: the program then has a deterministic semantics given by elaboration.

But what does it mean for programs be equivalent?

On program equivalence

There are several notions of program equivalence:

- If programs have a denotational semantics, the equivalence of programs should be the equality of their denotations.
- As a subcase, two programs having a common reduct should definitely be equivalent. However, this will in general not be complete: values may contain functions that are not identical, but perhaps would reduce to the same value whenever applied to the same arguments.
- This leads to the notion of *zobservational equivalence*. Two expressions are observationally equivalent (at some observable type, such as integers) if their are indistinguishable whenever they are put in arbitrary (well-typed) contexts of the observable type.

On program equivalence

For instance, two different elaborations that would just consistently change the representation of dictionaries (*e.g.* by ordering records in reverse order), would be equivalent if we cannot observe the representation of dictionaries.

Sufficient conditions for coherence

Since terms are explicitly typed, the only source of non-determinism is the elaboration of dictionaries.

One way to ensure coherence is that two dictionary *values* of the same type are always equal. This does not mean that there is a unique way of building dictionaries, but that all ways are equivalent as they eventually return the same dictionary.

Elaboration of dictionaries

Elaboration of dictionaries is just typing in System F.

More precisely, it infers a dictionary q given Γ and Q so that $\Gamma \vdash q : Q$.

The relevant subset of rules for dictionary expressions are:

D-OVar
$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

D-Inst
$$\frac{\Gamma \vdash q : \forall \alpha. \sigma}{\Gamma \vdash q\ \tau : [\alpha \mapsto \tau]\sigma}$$

D-App
$$\frac{\Gamma \vdash q_1 : Q_1 \Rightarrow Q_2 \qquad \Gamma \vdash q_2 : Q_1}{\Gamma \vdash q_1\ q_2 : Q_2}$$

Can we give a type-directed presentation?

Elaboration of dictionaries

Elaboration is driven by the type of the expected dictionary and the bindings available in the typing environment, which may be:

- a dictionary constructor z_h given by an instance definition h;
- a dictionary accessor $u_K^{K'}$ given by a class declaration K';
- a dictionary argument z, given by the local typing context.

Hence, the typing rules may be reorganized as follows:

$$\textsc{D-OVar-Inst}\quad \frac{z_h : \forall \vec{\beta}.\, P_1 \Rightarrow \ldots\ P_n \Rightarrow K\,(G\,\vec{\beta}) \in \Gamma \qquad \Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i}{\Gamma \vdash z_h\ \vec{\tau}\ \vec{q} : K\,(G\,\vec{\tau})}$$

$$\textsc{D-Proj}\quad \frac{u_K^{K'} : \forall \alpha.\, K'\ \alpha \Rightarrow K\ \alpha \in \Gamma \qquad \Gamma \vdash q : K'\ \tau}{\Gamma \vdash u_K^{K'}\ \tau\ q : K\ \tau}$$

$$\textsc{D-Var}\quad \frac{z : K\ \alpha \in \Gamma}{\Gamma \vdash z : K\ \alpha}$$

Elaboration of dictionary *values*

Dictionary values are typed in Γ_0, which does not contain free type variables, hence, the last rule does not apply.

Dictionary stored in other dictionaries must have been built in the first place. Hence, all dictionary values can be built with the *unique* rule:

$$\frac{\begin{array}{cc} z_h : \forall \vec{\beta}.\, P_1 \Rightarrow \ldots\ P_n \Rightarrow K\,(G\,\vec{\beta}) \in \Gamma & \Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i \end{array}}{\Gamma \vdash z_h\ \vec{\tau}\ \vec{q} : K\,(G\,\vec{\tau})}\ \textsc{D-OVar-Inst}$$

This rule for the judgment $\Gamma \vdash q : \tau$ can be read as an algorithm where Γ and τ are inputs (and Γ is constant) and q is an output.

There is no choice in finding $z_h : \forall \vec{\beta}.\, P_1 \Rightarrow \ldots\ P_n \Rightarrow K\,(G\,\vec{\beta}) \in \Gamma$, since each such clause is coming from an instance definition h, and we requested that *instance definitions never overlap*.

This ensures *uniqueness of dictionary* **values**, hence *coherence*.

Overlapping instances

Two instances *inst* $\forall \vec{\beta}_i .\ \vec{P} \Rightarrow K\,(G_i\ \vec{\beta}_i)\ \{r_i\}$ for i in $\{1, 2\}$ of a class K overlap if the type schemes $\forall \vec{\beta}_i .\ K\,(G_i\ \vec{\tau}_i)$ have a common instance, *i.e.* in the present setting, if G_1 and G_2 are equal.

Overlapping instances are an inherent source of incoherence: it means that for some type Q (in the common instance), a dictionary of type Q may (possibly) be built using two different implementations.

Elaboration of dictionary *arguments*

Dictionary expressions, as opposed to dictionary values, will also be built by extracting dictionaries from other dictionaries.

Why? Indeed, in overloaded code, the exact type is not fully known at compile time, hence dictionaries must be passed as arguments, from which superclass dictionaries may (and must, as we forbid to pass both a class and one of its super class dictionary simultaneously) be extracted.

Technically, they are typed in an extension of the typing context Γ_0 which may contain typing assumptions $z : K' \; \beta$ about dictionaries received as arguments. Hence rules D-Proj and D-Var may also apply.

Elaboration of dictionary arguments

The elaboration of dictionaries uses the three rules (reminder):

$$\text{D-OVAR-INST}\quad \frac{z : \forall \vec{\beta}.\, P_1 \Rightarrow \ldots\, P_n \Rightarrow K\,(G\,\vec{\beta}) \in \Gamma \qquad \Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i}{\Gamma \vdash z\,\vec{\tau}\,\vec{q} : K\,(G\,\vec{\tau})}$$

$$\text{D-PROJ}\quad \frac{u : \forall \alpha.\, K'\,\alpha \Rightarrow K\,\alpha \in \Gamma \qquad \Gamma \vdash q : K'\,\tau}{\Gamma \vdash z\,\tau\,q : K\,\tau} \qquad\qquad \text{D-VAR}\quad \frac{z : K\,\alpha \in \Gamma}{\Gamma \vdash z : K\,\alpha}$$

They can be read as a prolog-like *backtracking* algorithm.

Elaboration of dictionary arguments — Termination

The proof search always terminates, since premises have smaller Q than the conclusion when using the lexicographic order of first the height of τ, then the reverse order of class inheritance:

- If no rule applies, we fail.
- If rule D-Var applies, the derivation ends with success.
- If rule D-Proj applies, the premise is called with a smaller problem since the height is unchanged and $K' \; \vec{\tau}$ with $K' \prec K$.
- If D-Ovar-Inst applies, the premises are called at type $K_i \; \tau_j$ where τ_j is subtype (i.e. subterm) of $\vec{\tau}$, hence of a strictly smaller height.

Elaboration of dictionary arguments — Non determinism

For instance, in the introduction, we defined two instances eqInt and ordInt, while the later contains an instance of the former.

Hence, a dictionary of type eqInt may be obtained:

- directly as *EqInt*, or
- indirectly as *OrdInt*.*Eq*, by projecting the *Eq* sub-dictionary of class *Ord* Int

In fact, the latter choice could then be reduced at compile time and be equivalent to the first one.

One may enforce determinism by fixing a simple and sensible strategy for elaboration. Restrict the use of rule D-Proj to cases where Q is P–when D-OVar-Inst does not apply. However, the extra flexibility is harmless and perhaps useful freedom for the compiler.

Typing dictionaries — Example

In the introductory example Γ_0 is:

$$
\begin{array}{rcll}
\mathsf{equal} & \triangleq & u_{\mathit{equal}} & : \forall\alpha.\, \mathit{Eq}\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \mathit{bool}, \\
\mathit{EqInt} & \triangleq & z_{\mathit{Eq}}^{\mathit{Int}} & : Eq\ \mathit{int} \\
\mathit{EqList} & \triangleq & z_{\mathit{Eq}}^{\mathit{List}} & : \forall\alpha.\, \mathit{Eq}\ \alpha \Rightarrow Eq\ (\mathit{list}\ \alpha) \\
\\
\mathit{EqOrd} & \triangleq & u_{\mathit{Eq}}^{\mathit{Ord}} & : \forall\alpha.\, \mathit{Ord}\ \alpha \Rightarrow \mathit{Eq}\ \alpha \\
\mathsf{lt} & \triangleq & u_{\mathit{lt}} & : \forall\alpha.\, \mathit{Ord}\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \mathit{bool}
\end{array}
$$

When elaborating the body of leq, we have to infer a dictionary for *EqOrd* X *OrdX* in the local context X, *OrdX* : *Ord* X. Thus, Γ is $\Gamma_0, \alpha, z : Ord\ \alpha$ and *EqOrd* is u_{Eq}^{Ord}. We have:

$$
\textsc{D-Proj}\ \dfrac{\overset{\textsc{D-OVar-Inst}}{\Gamma \vdash u_{\mathit{Eq}}^{\mathit{Ord}}\ \alpha : \mathit{Ord}\ \alpha \rightarrow \mathit{Eq}\ \alpha} \qquad \overset{\textsc{D-Var}}{\Gamma \vdash z : \mathit{Ord}\ \alpha}}{\Gamma \vdash u_{\mathit{Eq}}^{\mathit{Ord}}\ \alpha\ z : \mathit{Eq}\ \alpha}
$$

Contents

What can be left implicit?

Class declarations? must remain explicit:

- They define the structure of dictionaries: a record type definition and its accessors.
- They define the type scheme of overloaded symbols and the class they belong to.

The type of instance declarations? must also remain explicit:

- These are polymorphic recursive definitions, hence their types are mandatory.

However, all core language expressions (in instance declarations and the final one) can be left implicit, in particular dictionary applications, but also abstractions over unresolved dictionaries.

Example

$$
\begin{array}{l}
\textbf{class } \mathit{Eq}\ X\ \ \{\ \mathsf{equal} : X \to X \to \mathsf{Bool}\ \}\\
\textbf{inst } \mathit{Eq}\ \mathsf{Int}\ \ \{\ \mathsf{equal} = \mathsf{primEqInt}\ \}\\
\textbf{inst } \mathit{Eq}\ \mathsf{Char}\ \{\ \mathsf{equal} = \mathsf{primEqChar}\ \}\\
\textbf{inst } \Lambda(X)\ \mathit{Eq}\ X \Rightarrow \mathit{Eq}\ (\mathsf{List}\ (X))\\
\quad \{\ \mathsf{eq} = \lambda(l_1 : \mathsf{List}\ X)\ \lambda(l_2 : \mathsf{List}\ X)\ \mathsf{match}\ l_1,\ l_2\ \mathsf{with}\\
\qquad\qquad \mid\ [],[]\ \to \mathsf{true}\ \mid\ [],\ \_\ \mid\ [],\ \_\ \to \mathsf{false}\\
\qquad\qquad \mid\ h_1 :: t_1,\ h_2 :: t_2 \to \mathsf{eq}\ X\ h_1\ h_2\ \&\&\ \mathsf{eq}\ (\mathsf{List}\ X)\ t_1\ t_2\ \}
\end{array}
$$

$$
\begin{array}{l}
\textbf{class } \mathit{Eq}\ (X) \Rightarrow \mathit{Ord}\ (X)\ \{\ \mathsf{lt} : X \to X \to \mathsf{Bool}\ \}\\
\textbf{inst } \mathit{Ord}\ (\mathsf{Int})\ \{\ \mathsf{lt} = (<)\ \}
\end{array}
$$

$$
\begin{array}{l}
\textbf{let rec } \mathsf{leq} : \forall(X)\ \mathsf{Ord}\ X \Rightarrow X \to \mathsf{List}\ X \to \mathsf{Bool} =\\
\quad \Lambda(X)\ \lambda(x : X)\ \lambda(l : \mathsf{List}\ X)\\
\qquad \mathsf{match}\ l\ \mathsf{with}\ []\ \to \mathsf{true}\\
\qquad \mid\ h :: t \to (\mathsf{equal}\ X\ x\ h\ \|\ \mathsf{lt}\ X\ x\ h)\ \&\&\ \mathsf{leq}\ X\ x\ t
\end{array}
$$

$$
\textbf{let } b = \mathsf{leq}\ \mathsf{Int}\ 1\ [1;\ 2;\ 3];;
$$

Type inference

The idea is to see dictionary types $K\,\tau$, which can only appear in type schemes and not in types, as a type constraint to mean *"there exists a dictionary of type $K\,\alpha$"*.

Just read $\forall\vec{\alpha}.\,\vec{P} \Rightarrow \tau$ as the constraint type scheme $\forall\vec{\alpha}[\vec{P}].\,\tau$.

We extend constraints with dictionary predicates:

$$C ::= \ldots \mid K\,\tau$$

On ground types a constraint $K\,\mathbf{t}$ is satisfied if one can build a dictionary of type $K\,\mathbf{t}$ in the initial environment Γ_0 (that contains all class and instance declarations), *i.e.* formally, if there exists a dictionary expression q such that $\Gamma_0 \vdash q : K\,\mathbf{t}$.

The satisfiability of class-membership constraints is thus:

$$\frac{K\,\phi\tau}{\phi \vdash K\,\tau}$$

Reasoning with class-membership constraints

For every class declaration *class* $K_1\ \alpha, \dots K_n\ \alpha \Rightarrow K\,\alpha\ \{\rho\}$,

$$K\,\alpha \ \Vdash\ K_1\ \alpha \wedge \dots\ K_n\ \alpha \qquad (1)$$

This rule allows to decompose any set of simple constraints into a canonical one.

Proof of (1). Assume $\phi \vdash K\,\alpha$, i.e. $\Gamma_0 \vdash q : K\,(\phi\alpha)$ *for some* q.

From the class declaration, we know that $K\,\alpha$ *is a record type definition that contains fields* $u^K_{K_i}$ *of type* $K_i\ \alpha_i$. *Hence, the dictionary value* q *contains field values of types* $K_i\ (\phi\alpha)$. *Therefore, we have* $\phi \vdash K_i\ \alpha$ *for all* i *in* $1..n$, *which implies* $\phi \vdash K_1\ \alpha \wedge \dots\ K_n\ \alpha$. □

Reasoning with class-membership constraints

For every instance definition *inst* $\forall \vec{\beta}.\ K_1\ \beta_1, \ldots K_p\ \beta_p \Rightarrow K\ (G\ \beta)\ \{r\}$

$$K\ (G\ \vec{\beta}) \quad \equiv \quad K_1\ \beta_1 \wedge \ldots\ K_p\ \beta_p \tag{2}$$

This rule allows to decompose all class constraints into simple constraints of the form $K\ \alpha$.

Proof of (2) ($\dashv$ direction).Assume $\phi \vdash K_i\ \beta_i$. There exists dictionaries q_i such that $\Gamma_0 \vdash q_i : K_i\ (\phi\beta_i)$. Hence, $\Gamma_0 \vdash x_h\ \vec{\beta}\ q_1\ \ldots\ q_p : K\ (G\ (\phi\vec{\beta}))$, i.e. $\phi \vdash K\ (G\ (\phi\vec{\beta}))$.

($\Vdash$ direction).Assume, $\phi \vdash K\ (G\ (\phi\vec{\beta})$. i.e. *there exists a dictionary q such that $\Gamma_0 \vdash q : K\ (G\ \phi\vec{\beta})$. By non-overlapping of instance declarations, the only way to build such a dictionary is by an application of x_h. Hence, q must be of the form $x_h\ \vec{\beta}\ q_1\ \ldots\ q_p$ with $\Gamma_0 \vdash q_i : K_i\ (\phi\beta_i)$, that is, $\phi \vdash K_i\ \beta_i$ for every i, which implies $\phi \vdash K_1\ \beta_1 \wedge \ldots\ K_p\ \beta_p$.*

Reasoning with class-membership constraints

For every instance definition *inst* $\forall\vec{\beta}.\, K_1\ \beta_1, \ldots K_p\ \beta_p \Rightarrow K\,(G\,\beta)\ \{r\}$

$$K\,(G\,\vec{\beta}) \quad \equiv \quad K_1\ \beta_1 \wedge \ldots\ K_p\ \beta_p \tag{2}$$

This rule allows to decompose all class constraints into simple constraints of the form $K\,\alpha$.

Notice that the equivalence still holds in an open-world assumption where new instance clauses may be added later, because another future instance definition cannot overlap with existing ones.

If overlapping of instances were allowed, the $\Vdash$ direction would not hold. Then, the rewriting rule:

$$K\,(G\,\vec{\beta}) \quad \longrightarrow \quad K_1\ \beta_1 \wedge \ldots\ K_p\ \beta_p$$

would still be sound (the right-hand side entails the left-hand side, and thus type inference would infer sound typings), i.e. but not complete (type inference could miss some typings).

Reasoning with class-membership constraints

For every class K and type constructor G for which there is no instance of K,

$$K\,(G\,\vec{\beta}) \quad \equiv \quad \mathit{false} \tag{3}$$

This rule allows failure to be reported as soon as constraints of the form $K\,(G\,\vec{\tau})$ appear and there is no instance of K for G.

Proof of (3).The $\dashv\!\Vdash$ direction is a tautology, so it suffices to prove the $\Vdash$ direction. By contradiction. Assume $\phi \vdash K\,(G\,\vec{\beta})$. This implies the existence of a dictionary q such that $\Gamma_0 \vdash q : K\,(G\,(\phi\vec{\beta}))$. Then, there must be some x_h in Γ whose type scheme is of the form $\forall\vec{\beta}.\vec{P} \Rightarrow K\,(G\,\vec{\beta})$, i.e. *there must be an instance of class K for G.*

Notice that this rule does not work in an open world assumption. The rewriting rule

$$K\,(G\,\vec{\beta}) \quad \longrightarrow \quad \mathit{false}$$

would still remain sound but incomplete.

Typing constraints

Constraint generation is as in ML.

A constraint type scheme can always be decomposed into one of the form $\forall\bar\alpha[P_1 \wedge P_2].\,\tau$ where $\mathrm{ftv}(P_1) \in \bar\alpha$ and $\mathrm{ftv}(P_2) \mathrel{\#} \bar\alpha$.

The constraints P_2 can then be extruded in the enclosing context if any, so we are in general left with just P_1.

Checking well-typedness

To check well-typedness of the program p equal to $\vec{H}\ \vec{h}\ a$, we must check that: each expression a_i^h and the expression a are well-typed, in the environment used to elaborate them:

This amounts to checking:

- $\Gamma_0, \Gamma_h \vdash a_i^h : \tau_i^h$ where τ_i^h is given.
  Thus, we check that the constraints *def* Γ_0, Γ_h *in* $(\!|a_i^h|\!) \preceq \tau_i^h \equiv$ *true*.
- $\Gamma_0 \vdash a : \tau$ for some τ.
  Thus, we check that *def* Γ_0 *in* $\exists\alpha. (\!|a|\!) \preceq \alpha \equiv$ *true*.

However, . . . Typechecking is not sufficient!

Type reconstruction should also return an explicitly-typed term M that can then be elaborated into N.

Type reconstruction

As for ML the resolution strategy for constraints may be tuned to keep persistent constraints from which an explicitly typed term M can be read back.

Back to coherence

When the source language is implicitly-typed, the elaboration from the source language into System F code is the composition of type reconstruction with elaboration of explicitly-typed terms.

That is, a elaborates to N if $\Gamma \vdash a \rightsquigarrow M : \tau$ and $\Gamma \vdash M \rightsquigarrow N : \tau$.

Hence, even if the elaboration is coherent for explicitly-typed terms, this may not be true for implicitly-typed terms.

There are two potential problems:

- The language has principal constrained type schemes, but the elaboration requests unconstrained type schemes.
- Ambiguities could be hidden (and missed) by non principal type reconstruction.

Coherence — Toplevel unresolved constraints

Thanks to the several restrictions on class declarations and instance definitions, the type system has principal constrained schemes (and principal typing reconstruction). However, this does not imply that there are principal *unconstrained* type schemes.

Indeed, assume that the principal constrained type scheme is $\forall\alpha[K\,\alpha].\,\alpha \to \alpha$ and the typing environment contains two instances of $K\,G_1$ and $K\,G_2$ of class K. Constraint-free instances of this type scheme are $G_1 \to G_1$ and $G_2 \to G_2$ but $\forall\alpha.\,\alpha \to \alpha$ is certainly not one.

Not only neither choice is principal, but the two choices would elaborate into expressions with different (non-equivalent) semantics.

We must fail in such cases.

Coherence

Toplevel unresolved constraints

This problem may appear while typechecking the final expression a in Γ_0 that request an unconstrained type scheme $\forall \alpha. \tau$

It may also occur when typechecking the body of an instance definition, which requests an explicit type scheme $\forall \vec{\alpha}[\vec{Q}]. \tau$ in Γ_0 or equivalently that requests a type τ in $\Gamma_0, \vec{\alpha}, \vec{Q}$.

Coherence

Example of unresolved constraints

class *Num* (X) { 0 : X, (+) : X → X → X }
inst *Num* Int { 0 = Int.(0), (+) = Int.(+} }
inst *Num* Float { 0 = Float.(0), (+) = Float.(+} }
let zero = 0 + 0;

The type of zero or zero + zero is $\forall\alpha[\mathit{Num}\ \alpha].\alpha$—and several classes are possible for $Num\ X$. The semantics of the program is undetermined.

class *Readable* (X) { read : descr → X }
inst *Readable* (Int) { read = read_int }
inst *Readable* (Char) { read = read_char }
let x = read (open_in())

The type of x is $\forall\alpha[\mathit{Readable}\ \alpha].\mathit{unit} \to \alpha$—and several classes are possible for *Readable* α. The program is rejected.

Coherence Inaccessible constraint variables

In the previous examples, the incoherence comes from the obligation to infer type schemes without constraints. A similar problem may occur with isolated constraints in a type scheme.

Assume, for instance, that the elaboration of $\mathit{let}\ x = a_1\ \mathit{in}\ a_2$ is $\mathit{let}\ x : \forall\alpha[K\,\alpha].\,\mathit{int} \to \mathit{int} = N_1\ \mathit{in}\ N_2$.

All applications of x in N_2 will lead to an unresolved constraint $K\,\alpha$ since neither the argument nor the context of this application can determine the value of the type parameter α. Still, a dictionary of type $K\,\tau$ must be given before N_1 can be executed.

Although x may not be used in N_2, in which case, all elaborations of the expression may be coherent, we may still raise an error, since an unusable local definition is certainly useless, hence probably a programmer's mistake. The error may then be raised immediately, at the definition site, instead of at every use of x.

Coherence — The open-world view

When there is a single instance $K\,G$ for a class K that appears in an unresolved or isolated constraint $K\,\alpha$, the problem formally disappears, as all possible type reconstructions are coherent.

However, we may still not accept this situation, for modularity reasons, as an extension of the program with another non-overlapping *correct* instance declaration would make the program become ambiguous.

Formally, this amounts to saying that the program must be coherent in its current form, but also in all possible extensions with well-typed class definitions. This is taking an *open-world* view.

On the importance of principal type reconstruction

In the source of incoherence we have seen, some class constraints remained undetermined.

As noticed earlier, some (usually arbitrary) less general elaboration would solve the problem—but the source program would remain incoherent.

Hence, in order to detect incoherent (i.e. ambiguous) programs it is essential that type reconstruction is principal.

Once a program has been checked coherent, *i.e.* with no undetermined constraint, based on a principal type reconstruction, can we still use another non principal type reconstruction for its elaboration?

Yes, indeed, this will preserve the semantics.

This freedom may actually be very useful for optimizations.

On the importance of principal type reconstruction

Consider the program

$$\textbf{let}\ \mathsf{twice} = \lambda(\mathsf{x})\ \mathsf{x} + \mathsf{x}\ \textbf{in}\ \mathsf{twice}\ (\mathsf{twice}\ 1)$$

Its principal type reconstruction is:

$$\textbf{let}\ \mathsf{twice} : \forall(\mathsf{X})\ [\ \mathit{Num}\ \mathsf{X}\ ]\ \mathsf{X} \to \mathsf{X} = \Lambda(\mathsf{X})\ [\mathit{Num}\ \mathsf{X}]\ \lambda(\mathsf{x})\ \mathsf{x} + \mathsf{x}\ \textbf{in}$$
$$\mathsf{twice}\ \mathsf{Int}\ (\mathsf{twice}\ \mathsf{Int})\ 1$$

which elaborates into

$$\textbf{let}\ \mathsf{twice}\ \mathsf{X}\ \mathsf{numX} = \lambda(\mathsf{x} : \mathsf{X})\ \mathsf{x}\ (\mathsf{plus}\ \mathsf{numX})\ \mathsf{x}\ \textbf{in}$$
$$\mathsf{twice}\ \mathsf{Int}\ \mathit{NumInt}\ (\mathsf{twice}\ \mathsf{Int}\ \mathit{NumInt}\ 1)$$

while, avoiding polyorphism, twice would elaborate into:

$$\textbf{let}\ \mathsf{twice} = \lambda(\mathsf{x} : \mathsf{Int})\ \mathsf{x}\ (\mathsf{plus}\ \mathit{NumInt})\ \mathsf{x}\ \textbf{in}\ \mathsf{twice}\ (\mathsf{twice}\ 1)$$

where moreover, the plus *NumInt* can be statically reduced.

Overloading by return types

All previous ambiguous examples are overloaded by return types:

- $0 : X$.
  The value 0 has an overloaded type that is not constraint by the argument.
- $\mathsf{read} : \mathsf{desc} \rightarrow X$.
  The function read applied to some ground type argument will be under specified.

Odersky et al. [1995] suggested to prevent overloading by return types by requesting that overloaded symbols of a class $K\,\alpha$ have types of the form $\alpha \rightarrow \tau$.

The above examples are indeed rejected by this definition.

Overloading by return types

In fact, disallowing overloading by return types suffices to ensure that all well-typed programs are coherent.

Moreover, untyped programs can then be given a semantics directly (which of course coincides with the semantics obtained by elaboration).

Many interesting examples of overloading fits in this schema.

However, overloading by returns types is also found useful in several cases and Haskell allows it, using default rules to resolve ambiguities.

This is still an arguable design choice in the Haskell community.

Contents

Changing the representation of dictionaries

An overloaded method call u of a class K is elaborated into an application $u\,q$ of u to a dictionary expression q of class K. The function u and the representation of the dictionary are both defined in the elaboration of the class K and need not be known at the call site.

This leaves quite a lot of flexibility in the representation of dictionaries.

For example, we used record data-type definitions to represent dictionaries, but tuples would have been sufficient.

An alternative compilation of type classes

The dictionary passing semantics is quite intuitive and very easy to type in the target language.

However, dictionaries may be replaced by a derivation tree that proves the existence of the dictionary. This derivation tree can be passed around instead of the dictionary and be used at the call site to dispatch to the appropriate implementation of the method.

This has been studied in [Furuse, 2003].

This can also elegantly be explained as a type preserving compilation of dictionaries called concretization and described in [Pottier and Gauthier, 2006]. It is somehow similar to defunctionalization and also requires that the target language be equipped with GADT (Guarded Abstract Data Types).

Multi-parameter type classes

Multi-parameter type classes are of the form

$$\mathit{class}\ \vec{P} \Rightarrow K\,\vec{\alpha}\ \{\rho\}$$

where free variables of $\vec{P}$ are in $\vec{\alpha}$.

The current framework can easily be extended to handle multi-parameter type classes.

Example: Collections represented by type C whose elements are of type E can be defined as follows:

```
class Collection C E { find : C → E → Option(E), add : C → E → C }
inst Collection (List X) X { find = List.find, add = λ(c)λ(e) e::c }
inst Collection (Set X) X { ... }
```

Type dependencies

However, the class Collection does not provide the intended intuition that collections be homogeneous:

let add2 c x y = add (add c x) y
add2 : ∀(C, E, E')
 Collection C E, Collection C E' ⇒ C → E → E' → C

This definition assumes that collections may be heterogeneous. This may not be intended, and perhaps no instance of heterogeneous collections will ever be provided.

To statically enforce collections to be homogeneous in types, the definition can add a clause to say that the parameter C determines the parameter E:

class Collection C E | C → E { ... }

Then, add2 would enforce E and E' to be equal.

Type dependencies

Type dependencies also reduce overlapping between class declarations.

Hence they allow examples that would have to be rejected if type dependencies could not be expressed.

Associated types

Functional dependencies are being replaced by the notion of *associated types*, which allows a class to declare its *own type function*.

Correspondingly, instance definitions must provide a definition for associated types (in addition to values for overloaded symbols).

For example, the Collection class becomes a single parameter class with an associated type definition:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Collection}\ \mathsf{E}\ \{ \\
\quad \textbf{type}\ \mathsf{C} : * \\
\quad \mathsf{find} : \mathsf{C} \to \mathsf{E} \to \mathsf{Option}\ \mathsf{E} \\
\quad \mathsf{add} : \mathsf{C} \to \mathsf{E} \to \mathsf{C} \\
\} \\
\textbf{inst}\ \mathsf{Collection}\ \mathit{Eq}\ \mathsf{X} \Rightarrow \mathsf{Collection}\ \mathsf{X}\ \{\textbf{type}\ \mathsf{C} = \mathsf{List}\ \mathsf{E}, \ldots\} \\
\textbf{inst}\ \mathsf{Collection}\ \mathit{Eq}\ \mathsf{X} \Rightarrow \mathsf{Collection}\ \mathsf{X}\ \{\textbf{type}\ \mathsf{C} = \mathsf{Set}\ \mathsf{E}, \ldots\}
\end{array}
$$

Associated types increase the expressiveness of type classes.

Overlapping instances

Example

In practice, overlapping instances may be desired! For example, one could provide a generic implementation of sets provided an ordering relation on elements, but also provide a more efficient version for bit sets.

If overlapping instances are allowed, further rules are needed to disambiguate the resolution of overloading, such as giving priority to rules, or using the most specific match.

However, the semantics depend on some particular resolution strategy and becomes more fragile. See [Jones et al., 1997] for a discussion.

See also [Morris and Jones, 2010] for a recent new proposal.

Overlapping instances

Example

inst *Eq*(X) { equal = (=) }
inst *Eq*(Int) { equal = primEqInt }

This elaborates into the creation of a generic dictionary

let *Eq* X : *Eq* X= { equal = (=) }
let *EqInt* : *Eq* Int = { equal = primEqInt }

Then, *EqInt* or *Eq* Int are two dictionaries of type *Eq* Int but with different implementations.

Restriction that are harder to lift

One may consider removing other restrictions on the class declarations or instance definitions. While some of these generalizations would make sense in theory, they may raise serious difficulties in practice.

For example:

- If constrained type schemes are of the form $K\,\tau$ instead of $K\,\alpha$? (which affects all aspects of the language), then it becomes difficult to control the termination of constrained resolution and of the elaboration of dictionaries.
- If a class instance *inst* $\forall \vec{\beta}.\, \vec{P} \Rightarrow K\,\tau\;\{\rho\}$ could be such that τ is $G\,\vec{\tau}$ and not $G\,\vec{\beta}$, then it would be more difficult to check non-overlapping of class instances.

Methods as overloading functions

One approach to object-orientation is to see methods as over as overloaded functions.

In this view, objects carry class tags that can be used at runtime to find the best matching definition.

This approach has been studied in detail by [Millstein and Chambers, 1999]. See also [Bonniot, 2002, 2005].

Implicit arguments

Oliviera et al noticed that type classes could be largely emulated in Scala with implicit arguments [Oliveira et al., 2010].

This has recently be formalized by Oliviera et al in COCHIS, a calculus with implicits arguments [Schrijvers et al., 2017].

This allows *local scoping* of overloaded functions, but coherence is solved by a restriction to first-choice commitment during resolution to force it to be deterministic.

Modules-based type classes

Modular type classes [Dreyer et al., 2007] mimic type classes at the level of modules, but with explicit abstraction and instantiations.

Modular implicits [White et al., 2014] allows for implicit module arguments. This extends the idea of modular type-classes in two directions.

- The language is more expressive
- Module arguments are inferred (left implicit).
- Module abstractions remain explicit. This allows for local scoping of overloading.

```
module type EQ =
  sig type t val eq : t → t → bool end

implicit module Eq_Int = struct
  type t = int let eq (x:int) y = x = y
end

implicit module Eq_Char = struct
  type t = char
  let eq (x:char) y = x = y end

let eq {Eq : EQ} x = Eq.eq x

implicit module Eq_List {Eq : EQ} =
  struct module rec R : EQ
      with type t = Eq.t list = struct
  type t = Eq.t list
  let eq l1 l2 = match l1, l2 with
   | [],[] → true | [], _ | _,[] → false
   | h1::t1, h2::t2 →
       eq h1 h2 && eq {R} t1 t2
end include R end
```

```
module type ORD = sig type t
  module Eq : EQ with type t = t
  val lt : t → t → bool end

implicit module Ord_Int = struct
  type t = int
  module Eq = Eq_Int
  let lt x y = (x < y) end

implicit module Eq {Ord : ORD} = Ord.Eq

let lt {Ord : ORD} x = Ord.lt x;;

implicit module Ord_List {Ord : ORD} =
  struct module rec R : ORD
      with type t = Ord.t list = struct
  type t = Ord.t list
  module Eq = Eq_List {Ord.Eq}
  let lt l1 l2 = match l1, l2 with
   | _, [] → false | [], _ → true
   | h1::t1, h2::t2 → lt h1 h2 && lt{R} t1 t2
end include R end
```

```
let leq (type a) {Ord : ORD with type t = a list} (l1 : a list) l2 =
    lt l1 l2 || eq l1 l2
```

Summary

Static overloading is not a solution for polymorphic languages. Dynamics overloading must be used instead.

Dynamics overloading is a powerful mechanism.

Haskell type classes are a practical, general, and powerful solution to dynamic overloading,

Dynamic overloading works relatively well in combination with ML-like type inference.

However, besides the simplest case where every one agrees, useful extensions often come with some drawbacks, and there is not yet an agreement on the best design choices.

The design decisions are often in favor of expressiveness, but loosing some of the properties and the canonicity of the minimal design.

Summary

Dynamic overloading is a typical and very elegant use of elaboration.

The programmer could in principle write the elaborated program, build and pass dictionaries explicitly, but this would be cumbersome, tricky, error prone, and it would obfuscate the code.

The elaboration does this automatically, without arbitrary choices (in the minimal design) and with only local transformations that preserve the structure of the source.

Further Reading

For an all-in-one explanation of Haskell-like overloading, see *The essence of Haskell* by Odersky et al.

See also the Jones's monograph *Qualified types: theory and practice*.

For a calculus of overloading see ML& [Castagna, 1997]

Type classes have also been added to Coq [Sozeau and Oury, 2008]. Interestingly, the elaboration of proof terms need not be coherent which makes it a simpler situation for overloading.

A technique similar to defunctionalization can be used to replace dictionnaries by tags, which are interpreted when calling an overloaded function to dispatch to the appropriate definition [Pottier and Gauthier, 2004].

Implicit module arguments [White et al., 2014] can also mimick type-classes overloading with some drawbacks and advantages.

Bibliography I

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

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, March 1996.

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2–3):81–116, December 1995.

▷ Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *ACM International Conference on Functional Programming (ICFP)*, pages 157–168, September 2008.

▷ Robert Atkey. Relational parametricity for higher kinds. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 46–61, 2012. doi: 10.4230/LIPIcs.CSL.2012.46.

Bibliography II

▷ Nick Benton and Andrew Kennedy. Exceptional syntax journal of functional programming. *J. Funct. Program.*, 11(4):395–410, 2001.

▷ Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. *Testing Polymorphic Properties*, pages 125–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-11957-6. doi: $10.1007/978\text{-}\sigma_2 3\text{-}\sigma_2 642\text{-}\sigma_2 11957\text{-}\sigma_2 6\_8$.

▷ Richard Bird and Lambert Meertens. Nested datatypes. In *International Conference on Mathematics of Program Construction (MPC)*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

▷ Clément Blaudeau. OCaml modules: formalization, insights and improvements. Master's thesis, École polytechnique fédérale de Lausanne, September 2021.

Daniel Bonniot. *Typage modulaire des multi-méthodes*. PhD thesis, École des Mines de Paris, November 2005.

▷ Daniel Bonniot. Type-checking multi-methods in ML (a modular approach). In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 2002.

Bibliography III

- Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33:309–338, 1998.
- Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.
- Yufei Cai, Paolo G. Giarrusso, and Klaus Ostermann. System F-omega with equirecursive types for datatype-generic programming. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 30–43. ACM, 2016. doi: 10.1145/2837614.2837660.

Luca Cardelli. An implementation of f¡:. Technical report, DEC Systems Research Center, 1993.

Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.

Bibliography IV

- Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. The MLton compiler, 2007.
- Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.
- Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 38–49, January 2005.
- Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 54–65, June 2007.
- Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

Julien Cretin and Didier Rémy. System F with Coercion Constraints. In *Logics In Computer Science (LICS)*. ACM, July 2014.

Bibliography V

▷ Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–70, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4.

Joshua Dunfield. Greedy bidirectional polymorphism. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 15–26, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-509-3. doi: http://doi.acm.org/10.1145/1596627.1596631.

▷ Jun Furuse. Extensional polymorphism by flow graph dispatching. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science*. Springer, November 2003.

▷ Jacques Garrigue. Relaxing the value restriction. In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, April 2004.

Bibliography VI

Jacques Garrigue and Didier Rémy. Ambivalent Types for Principal Type Inference with GADTs. In *11th Asian Symposium on Programming Languages and Systems*, Melbourne, Australia, December 2013.

▷ Nadji Gauthier and François Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 150–161, September 2004. doi: http://doi.acm.org/10.1145/1016850.1016872.

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

▷ Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, 1990.

Bibliography VII

▷ Neal Glew. A theory of second-order trees. In Daniel Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2002. doi: $10.1007/3\text{-}\sigma_2 540\text{-}\sigma_2 45927\text{-}\sigma_2 8\backslash\_11$.

Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, 2012.

Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–345. MIT Press, 2005.

▷ Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

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

Bibliography VIII

J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

▷ Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *ACM SIGPLAN Conference on History of Programming Languages*, June 2007.

Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris 7, September 1976.

Mark P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995. ISBN 0-521-47253-9.

▷ Mark P. Jones. Typing Haskell in Haskell. In *Haskell workshop*, October 1999.

▷ Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, 1997.

Bibliography IX

Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 193–204, New York, NY, USA, 1992. ACM. ISBN 0-89791-481-3. doi: http://doi.acm.org/10.1145/141471.141540.

▷ Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, pages 206–220. Springer, May 1990.

▷ Oleg Kiselyov. Higher-kinded bounded polymorphism. web page.

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

▷ Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16 (5):1411–1430, September 1994.

Bibliography X

▷ Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of system F. In *ACM International Conference on Functional Programming (ICFP)*, pages 27–38, August 2003.

Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.

▷ Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris 7, June 1992.

▷ Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/349214.349230.

▷ John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–57, January 1988.

Bibliography XI

▷ Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 382–401, 1990.

▷ Sophie Malecki. Proofs in system fω can be done in system fω1. In Dirk van Dalen and Marc Bezem, editors, *Computer Science Logic*, pages 297–315, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69201-0.

Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17, May 2007.

▷ David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer, June 2003.

Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 279–303, London, UK, 1999. Springer-Verlag. ISBN 3-540-66156-5.

Bibliography XII

- Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.
- Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–283, January 1996.
- John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2–3):211–249, 1988.
- John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.
- Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 63–74, January 2009.

Bibliography XIII

J. Garrett Morris and Mark P. Jones. Instance chains: type class programming without overlapping instances. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 375–386, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: http://doi.acm.org/10.1145/1863543.1863596.

▷ Greg Morrisett and Robert Harper. Typed closure conversion for recursively-defined functions (extended abstract). In *International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

▷ Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

▷ Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, April 1984.

Bibliography XIV

- Hiroshi Nakano. A modality for recursion. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 255–266, June 2000.
- Hiroshi Nakano. Fixed-point logic with the approximation modality and its Kripke completeness. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, volume 2215 of *Lecture Notes in Computer Science*, pages 165–182. Springer, October 2001.
- Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 135–146, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7.
- Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
- Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 41–53, 2001.

Bibliography XV

- Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1999.
- Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869489.
- Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Online lecture notes, January 2009.
- Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Manuscript, April 2004.
- Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, January 1993.

Bibliography XVI

Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: http://doi.acm.org/10.1145/62678.62697.

Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

▷ Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

▷ Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

▷ François Pottier. Notes du cours de DEA "Typage et Programmation", December 2002.

François Pottier. A typed store-passing translation for general references. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, Austin, Texas, January 2011. Supplementary material.

Bibliography XVII

François Pottier. Hindley-Milner elaboration in applicative style. In *Proceedings of the 2014 ACM SIGPLAN International Conference on Functional Programming (ICFP'14)*, September 2014.

▷ François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 89–98, January 2004.

▷ François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. Submitted for publication, October 2012.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 173–184, September 2013.

Bibliography XVIII

- François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 232–244, January 2006.
- François Pottier and Yann Régis-Gianas. Towards efficient, typed LR parsers. In *ACM Workshop on ML*, volume 148-2 of *Electronic Notes in Theoretical Computer Science*, pages 155–180, March 2006.
- François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
- François Pottier and Didier Rémy. The essence of ML type inference. Draft of an extended version. Unpublished, September 2003.
- Jonathan Protzenko. *Mezzo, a typed language for safe effectful concurrent programs.* PhD thesis, University Paris Diderot, 2014.
- Didier Rémy. Simple, partial type-inference for System F based on type-containment. In *Proceedings of the tenth International Conference on Functional Programming*, September 2005.

Bibliography XIX

- Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346. Springer, April 1994.
- Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

  Didier Rémy and Boris Yakobowski. Efficient Type Inference for the MLF language: a graphical and constraints-based approach. In *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 63–74, Victoria, BC, Canada, September 2008. doi: http://doi.acm.org/10.1145/1411203.1411216.
- John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, April 1974.
- John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

Bibliography XX

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

▷ Andreas Rossberg. 1ml - core and modules united. *J. Funct. Program.*, 28:e22, 2018. doi: 10.1017/S0956796818000205.

▷ Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. *J. Funct. Program.*, 24(5):529–607, 2014. doi: 10.1017/S0956796814000264.

François Rouaix. Safe run-time overloading. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 355–366, 1990. doi: http://doi.acm.org/10.1145/96709.96746.

Bibliography XXI

- Gabriel Scherer and Didier Rémy. Full reduction in the face of absurdity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 685–709, 2015. doi: $10.1007/978\text{-}\sigma_2 3\text{-}\sigma_2 662\text{-}\sigma_2 46669\text{-}\sigma_2 8\_28$.
- Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. Cochis: Deterministic and coherent implicits. Technical report, KU Leuven, May 2017.
- Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Program. Lang. Syst.*, 29(1), January 2007. ISSN 0164-0925. doi: $10.1145/1180475.1180476$.
- Christian Skalka and François Pottier. Syntactic type soundness for HM(X). In *Workshop on Types in Programming (TIP)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, July 2002.

Bibliography XXII

- Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In Sofiène Tahar, Otmame Ait-Mohamed, and César Muñoz, editors, *TPHOLs 2008: Theorem Proving in Higher Order Logics, 21th International Conference*, Lecture Notes in Computer Science. Springer, August 2008.
- Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.
- Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000.
- Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM. ISBN 1-59593-393-X. doi: 10.1145/1190315.1190324.
- W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):pp. 198–212, 1967. ISSN 00224812.

Bibliography XXIII

▷ Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 11(2):245–296, 1994.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

▷ Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179(1):1–18, 2002.

▷ Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, September 2004.

▷ Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8 (4):367–412, July 1998.

▷ Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, September 2011. ISSN 0956-7968. doi: 10.1017/S0956796811000098.

Bibliography XXIV

- Philip Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, September 1989.
- Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1–3):201–226, May 2007.
- Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–76, January 1989.
- J. B. Wells. The essence of principal typings. In *International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer, 2002.
- J. B. Wells. The undecidability of Mitchell's subtyping relation. Technical Report 95-019, Computer Science Department, Boston University, December 1995.
- J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

Bibliography XXV

- Leo White, Frédéric Bour, and Jeremy Yallop. Modular implicits. In Oleg Kiselyov and Jacques Garrigue, editors, *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.*, volume 198 of *EPTCS*, pages 22–63, 2014. doi: 10.4204/EPTCS.198.2.
- Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.
- Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.
- Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 119–135, Cham, 2014. Springer International Publishing. ISBN 978-3-319-07151-0.