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

Didier Rémy

Fall, 2021

Plan of the course

Metatheory of System F

ADTs, Recursive types, Existential types, GATDs

Going higher order with F^ω!

Logical relations

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 is done in Coq, for your help or curiosity,
—but not (yet) mandatory.

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* (*e.g.* ML or System F),
- 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 split 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:

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

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

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*:

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

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

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*:

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

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* $\Gamma \vdash M_1 : \tau_2 \to \tau$ *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* $\tau_2 \to \tau_1$ *and* $\Gamma, x : \tau_2 \vdash M_1 : \tau_1$*.*

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?

?

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

On the one hand, type erasure preserves reduction.

$$M_1 \xrightarrow{\beta} M_2$$

Lemma (Direct simulation)

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

Untyped semantics

On the one hand, type erasure preserves reduction.

Lemma (Direct simulation)

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

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

Untyped semantics

On the one hand, 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.

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

Lemma (Inverse simulation)

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

Untyped semantics

On the one hand, 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* $\lceil M' \rceil = a$.

Untyped semantics

On the one hand, 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…

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 carried again, in 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:

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

Pairs

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

Values, expressions, evaluation contexts are extended:

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

A new reduction rule is introduced:

$$\boxed{???}$$

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:

???

Pairs

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

Values, expressions, evaluation contexts are extended:

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

A new reduction rule is introduced:

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

Pairs

Types are extended:

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

Two new typing rules are introduced:

$$\textsc{Pair}\quad \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2}$$

$$\textsc{Proj}\quad \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathit{proj}_i\ M : \tau_i}$$

Sums

Values, expressions, evaluation contexts are extended:

$$\begin{array}{rcl} M & ::= & \ldots \mid \mathit{inj}_i\ M \mid \mathit{case}\ M\ \mathit{of}\ V\ [\!]\ V \\ E & ::= & \ldots \mid\ ??? \\ V & ::= & \ldots \mid \mathit{inj}_i\ V \end{array}$$

A new reduction rule is introduced:

???

Sums

Values, expressions, evaluation contexts are extended:

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

A new reduction rule is introduced:

???

Sums

Values, expressions, evaluation contexts are extended:

$$\begin{array}{rcl} M & ::= & \ldots \mid \mathit{inj}_i\ M \mid \mathit{case}\ M\ \mathit{of}\ V \parallel V \\ E & ::= & \ldots \mid \mathit{inj}_i\ [] \mid \mathit{case}\ []\ \mathit{of}\ V \parallel V \\ V & ::= & \ldots \mid \mathit{inj}_i\ V \end{array}$$

A new reduction rule is introduced:

$$\mathit{case}\ \mathit{inj}_i\ V\ \mathit{of}\ V_1 \parallel V_2 \longrightarrow V_i\ V$$

Sums

Types are extended:

$$\tau ::= \dots \mid \tau + \tau$$

Two new typing rules are introduced:

$$\textsc{Inj}\ \frac{\Gamma \vdash M : \tau_i}{\Gamma \vdash \mathit{inj}_i\ M : \tau_1 + \tau_2} \qquad \textsc{Case}\ \frac{\Gamma \vdash M : \tau_1 + \tau_2 \qquad \Gamma \vdash V_1 : \tau_1 \to \tau \qquad \Gamma \vdash V_2 : \tau_2 \to \tau}{\Gamma \vdash \mathit{case}\ M\ \mathit{of}\ V_1 \parallel V_2 : \tau}$$

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

?

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:

?

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:

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

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 ...

?

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 . . .

?

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?

?

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.

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:

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

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:

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

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 "$(\lambda x{:}\tau.\, M_2)\ M_1$".

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*:

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

The construct "$M_1; M_2$" can in turn be viewed as syntactic sugar for ...

?

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*:

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

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.

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

This seems better...

?

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.

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

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.

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".

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:

$$(\top \to \top \to bool) \to list\ \top \to list\ \top$$

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

Why isn't this so good?

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.)

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

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 $\forall\alpha.\,\alpha \rightarrow \alpha$ has only a few inhabitants, which ones?

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.

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.

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!

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.

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

Can you give a few?

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.

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.

This phenomenon was studied by Reynolds [1983] and by Wadler [1989; 2007], among others. An account based on an operational semantics is offered by Pitts [2000].

Unfortunately, parametricity theorems are invalidated or degenerate in the presence of side effects (non-termination, exceptions, or references).

While most programs use side effects and side effects cannot be ignored when reasoning globally, many parts of programs do not use them and reasoning locally as if they where no side effects is still often helpful.

Parametricity plays an important role in the study of functional programming languages and remains a guideline when programming.

See the course on logical relations.

Ad hoc versus parametric

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:

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

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:

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

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:

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

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:

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

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:

$$\textsc{WfEnv-Empty}\quad \dfrac{}{\vdash \varnothing}$$

$$\textsc{WfEnvTvar}\quad \dfrac{\vdash \Gamma \qquad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha}$$

$$\textsc{WfEnvVar}\quad \dfrac{\Gamma \vdash \tau \qquad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : \tau}$$

$$\textsc{WfTypeVar}\quad \dfrac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha}$$

$$\textsc{WfTypeArrow}\quad \dfrac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2}$$

$$\textsc{WfTypeForall}\quad \dfrac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha.\, \tau}$$

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 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.

$$\text{VAR}\quad \frac{x : \tau \in \Gamma \qquad \boxed{\vdash \Gamma}}{\Gamma \vdash x : \tau} \qquad\qquad \text{TAPP}\quad \frac{\Gamma \vdash M : \forall \alpha.\tau \qquad \boxed{\Gamma \vdash \tau'}}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}$$

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.

?

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:

$$V ::= \dots \qquad\qquad E ::= \dots$$

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,

?

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:

$$?$$

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:

?

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:

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

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 $\forall\alpha.\, \mathit{list}\,(\mathit{list}\,\alpha) \to \mathit{list}\,\alpha$, the monomorphic function $(f\ \mathit{int}) \circ (f\ (\mathit{list}\ \mathit{int}))$ reduces to $\Lambda x.\, f\ (f\ x)$, while its more general polymorphic version $\Lambda\alpha.\,(f\ \alpha) \circ (f\ (\mathit{list}\,\alpha))$ 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

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \tau \to \tau \mid \forall \alpha.\tau \\ M & ::= & x \mid \lambda x{:}\tau.\, M \mid M\ M \mid \Lambda\alpha.\, M \mid M\ \tau \end{array}$$

Typing rules

Var
$$\Gamma \vdash x : \Gamma(x)$$

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

Tabs
$$\frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\tau}$$

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

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

Semantics

$$\begin{array}{lcl} V & ::= & \lambda x{:}\tau.\, M \mid \Lambda\alpha.\, V \\ E & ::= & [\,]\ M \mid V\ [\,] \mid [\,]\ \tau \mid \Lambda\alpha.\, [\,] \end{array}$$

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

$$(\Lambda\alpha.\, V)\ \tau \longrightarrow [\alpha \mapsto \tau]V$$

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

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 is 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:

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

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:

$$\begin{array}{rcll} V & ::= & \ldots & \mid Pair \mid Pair\ \tau \mid Pair\ \tau\ \tau \mid Pair\ \tau\ \tau\ V \mid Pair\ \tau\ \tau\ V\ V \\ & & & \mid proj_i \mid proj_i\ \tau \mid proj_i\ \tau\ \tau \end{array}$$

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?

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 $\{\mathit{unit}, \mathit{int}, \mathit{bool}, (\_ \times \_), \mathit{list}\ \_, \ldots\}$

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:

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

We require that

- $\boxed{\tau}$ be is 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:

$$M ::= \ldots \mid c \qquad c ::= C \mid d$$

$$\textsc{Cst}\quad \frac{c : \tau \in \Delta}{\Gamma \vdash c : \tau}$$

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

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

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 $d\ \tau_1\ \ldots\ \tau_k\ V_1\ \ldots\ V_{\text{arity}(d)}$ 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:

?

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,

?

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.

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)

Exercise

Lists

1) Formulate the extension of System F with lists as constants.
2) Check that this extension is sound.

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 substititition, 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

?

Compositionality

Lemma (Compositionality)

If $\Gamma \vdash E[M] : \tau$, then there exists $\vec{\alpha}$ and τ' such that $\Gamma, \vec{\alpha} \vdash M : \tau'$ and all M' verifying $\Gamma, \vec{\alpha} \vdash M' : \tau'$ also verify $\Gamma \vdash E[M'] : \tau$.

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 variable, 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*

?

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 a function*

?

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.*

?

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*

?

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*

?

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 into 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:

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

Notice that these rules are not syntax directed.

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.

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

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:

$$
\begin{tikzcd}
M_0 \arrow[rr, dashed, "*", "\iota"'] \arrow[dr, dotted] & & M_0' \arrow[r, "\beta\delta"'] \arrow[dl, dotted] & M_1 \arrow[d, dotted] \\
& a_0 \arrow[rr, "\beta\delta"'] & & a_1
\end{tikzcd}
$$

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:

$$
\begin{tikzcd}
M_0 \arrow[r, dashed, "*", "\iota"'] \arrow[d, dotted] & M'_0 \arrow[r, "\beta\delta"'] \arrow[dl, dotted] & M_1 \arrow[d, dotted] & & M_j \arrow[r, dashed, "*", "\iota"'] \arrow[d, dotted] & M'_j \arrow[r, "\beta\delta"'] \arrow[dl, dotted] & M_{j+1} \arrow[d, dotted] \\
a_0 \arrow[rr, "\beta\delta"'] & & a_1 & \dots & a_j \arrow[rr, "\beta\delta"'] & & a_{j+1}
\end{tikzcd}
$$

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:

$$
\begin{tikzcd}
M_0 \arrow[r, dashed, "*", "\iota"'] \arrow[d, dotted] & M_0' \arrow[r, "\beta\delta"'] \arrow[dl, dotted] & M_1 \arrow[d, dotted] & & M_j \arrow[r, dashed, "*", "\iota"'] \arrow[d, dotted] & M_j' \arrow[r, "\beta\delta"'] \arrow[dl, dotted] & M_{j+1} \arrow[d, dotted] & & M_n \arrow[r, dashed, "*", "\iota"'] \arrow[d, dotted] & V \arrow[r, "/" marking] \arrow[dl, dotted] & {} \\
a_0 \arrow[rr, "\beta\delta"'] & & a_1 & \dots & a_j \arrow[rr, "\beta\delta"'] & & a_{j+1} & \dots & a_n = v \arrow[r, "/" marking] & {}
\end{tikzcd}
$$

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 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.

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

(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. □

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 (implicit version for conciseness):

?

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 (implicit version for conciseness):

- a new type constructor G,

?

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 (implicit version for conciseness):

- a new type constructor G,
- n constructors $\quad C_i \;:\; \forall\vec\alpha.\,\tau_i \to G\,\vec\alpha$

?

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 (implicit version for conciseness):

- a new type constructor G,
- n constructors $C_i : \forall \vec{\alpha}.\, \tau_i \to G\,\vec{\alpha}$
- one destructor $d_G : \forall \vec{\alpha}, \gamma.\, G\,\vec{\alpha} \to (\tau_1 \to \gamma) \ldots (\tau_n \to \gamma) \to \gamma$

?

Algebraic Datatypes Types

General case

General case

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

In System F, this amounts to declaring (implicit version for conciseness):

- 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 $\quad d_G\,(C_i\;v)\;v_1\;\ldots v_n \;\rightsquigarrow\; v_i\;v$

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:

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

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

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.

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

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

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

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

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.)

Equirecursive types — Without quantifiers

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

By coinduction

Let $\begin{array}{|l} u \text{ be } \mu\alpha A A \alpha \\ v \text{ be } \mu\alpha A A A \alpha \end{array}$

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

Equirecursive types — Without quantifiers

Proof of $\mu\alpha AA\alpha = \mu\alpha AAA\alpha$

By coinduction

By unification

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

Equirecursive types Equality

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 gray, 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.

?

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 was is 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)?

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 *rec* $\alpha.\alpha \to \alpha$
- 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 iso- and) equirecursive types.

- equirecursive types are restricted by default to object or data types.
- 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:

?

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:

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

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

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

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

Exercise

What are the corresponding declarations in System F?

?

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,*

?

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* $\quad C_\Pi \;:\; \forall\vec{\alpha}.\,\tau_1 \to \dots \tau_n \to G\,\vec{\alpha}$

?

Records

A record can be defined as

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

Exercise

What are the corresponding declarations in System F?

- *a new type constructor* G_Π,
- 1 *constructor* $\quad C_\Pi : \forall\vec{\alpha}.\,\tau_1 \to \ldots \tau_n \to G\,\vec{\alpha}$
- n *destructors* $\quad d_{\ell_i} : \forall\vec{\alpha}.\,G\,\vec{\alpha} \to \tau_i$

?

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* $\quad C_\Pi \;:\; \forall \vec{\alpha}.\, \tau_1 \to \ldots \tau_n \to G\,\vec{\alpha}$
- n *destructors* $\quad d_{\ell_i} \;:\; \forall \vec{\alpha}.\, G\,\vec{\alpha} \to \tau_i$
- n *reduction rules* $\quad d_{\ell_i}(C_\Pi\; v_1\; \ldots v_n) \;\leadsto\; v_i$

?

Records

A record can be defined as

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

Exercise

What are the corresponding declarations in System F?

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

Can a record also be used for defining recursive types?
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]

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 with GADTs.)

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

Contents

Existential types *Examples*

A frozen application returning a value of type (≈ a thunk)

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

Existential types *Examples*

A frozen application returning a value of type (≈ a thunk)

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

Type of closures in the environment-passing variant:

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

Existential types *Examples*

A frozen application returning a value of type (≈ 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:

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

Existential types

One can extend System F with *existential types,* in addition to universals:

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

As in the case of universals, there are *type-passing* and *type-erasing* interpretations of the terms and typing rules... and in the latter interpretation, there are *explicit* and *implicit* versions.

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:

Pack

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

Unpack

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

Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

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

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

Anything wrong?

Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

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

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

The side condition $\alpha \mathrel{\#} \tau_2$ is mandatory here to ensure well-formedness of the conclusion.

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

Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

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

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

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?

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.

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

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:

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

On existential elimination

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

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

On existential elimination

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

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

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

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

On existential elimination

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

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

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

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

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

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

On existential elimination

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

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

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

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

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

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

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

On existential elimination

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

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

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

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

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

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

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

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

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:

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

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:*

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

The term translation is:

$$\begin{array}{rcl} [\![\mathit{pack}_{\exists\alpha.\tau}]\!] & : & \forall\alpha.\,([\![\tau]\!] \rightarrow [\![\exists\alpha.\tau]\!]) \\ & = & ? \\ [\![\mathit{unpack}_{\exists\alpha.\tau}]\!] & : & [\![\exists\alpha.\tau]\!] \rightarrow \forall\beta.\,((\forall\alpha.\,([\![\tau]\!] \rightarrow \beta)) \rightarrow \beta) \\ & = & ? \end{array}$$

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:

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

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:

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

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:

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

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:

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

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?

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:

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

We add the reduction rule:

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

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.*

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

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.*

Solution

Let M_1 be if M then V_1 else V_2 where V_i is of the form *pack* τ_i, V_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?

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:

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

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:

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

Is the semantics type-erasing?

Implicitly-typed existential types subtlety

Yes, it is.

But there is a subtlety!

Implicitly-typed existential types subtlety

Yes, it is.

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

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 $\lambda f.\,\lambda x.\,not\ (f\ (not\ x))$.

Let v be the function $\lambda b.\,\text{if } b \text{ then } v_1 \text{ else } v_2$ of type $\mathit{bool} \to \tau_0$.

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

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

What happened?

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 $\lambda f.\,\lambda x.\,not\ (f\ (not\ x))$.

Let v be the function $\lambda b.\,\text{if } b \text{ then } v_1 \text{ else } v_2$ of type $\textit{bool} \to \tau_0$.

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

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

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?

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 break.

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:*

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

This introduces two constants, with the following type schemes:

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

(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?

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:

$$\textsc{Unpack}_D\qquad \frac{\begin{array}{c}\Gamma \vdash M_1 : D\;\vec{\tau} \\ \Gamma \vdash M_2 : \forall \bar{\beta}.\,([\vec{\alpha} \mapsto \vec{\tau}]\tau \to \tau_2) \qquad \bar{\beta} \,\#\, \vec{\tau}, \tau_2\end{array}}{\Gamma \vdash \textit{unpack}_D\; M_1\; M_2 : \tau_2}$$

where $D\;\vec{\alpha} \approx \exists \bar{\beta}.\tau$

We have seen a version of this rule in System F earlier; this in an ML 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):

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

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:

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

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

Existential types calls for universal types!

Exercise We reuse the type $D\ \alpha \approx \exists\beta.(\beta \to \alpha) \times \beta$ of frozen computations. Assume given a list l with elements of type $D\ \tau_1$.

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 $D\ \tau_2$ (without actually running any computation).

List.map (λ(z) **let** D(f, y) = z **in** D((λ(z) g (f z)), y))

Try generalizing this example to a function that receives g and l and returns l'

?

Existential types calls for universal types!

Exercise We reuse the type $D\ \alpha \approx \exists \beta.(\beta \to \alpha) \times \beta$ of frozen computations. Assume given a list l with elements of type $D\ \tau_1$.

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 $D\ \tau_2$ (without actually running any computation).

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

?

Existential types calls for universal types!

Exercise We reuse the type $D\ \alpha \approx \exists\beta.(\beta \to \alpha) \times \beta$ of frozen computations. Assume given a list l with elements of type $D\ \tau_1$.

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 $D\ \tau_2$ (without actually running any computation).

```
List.map (λ(z) let D(f, y) = z in D((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 D(f, y) = z in D((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 so that x 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 Thunk (succ, 1) where Thunk is either one of

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

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

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 encoding 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 d = D : ('b → 'a) * 'b → 'a d
let freeze f x = D (f, x)
let unfreeze (D (f, x)) = f x
```

Via first-class modules (abstract types)

```
module type D = 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 : D)
let unfreeze (type u) (module M : D 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

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

GADTs

This is no more the case with GADTs

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

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

What are they?

ADTs

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

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

GADTs

This is no more the case with GADTs

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

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
```

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
```

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 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
```

Key point: the typechecker refines the types a and b in each branch

```
let rec apply : type a b. (a, b) apply → a → b = fun f arg →
  match f with                                                (* a =        b = *)
  | Fadd    → let x, y = arg in x + y                         (* int * int  int   *)
  | Fnot    → let x = arg in if x then false else true        (* bool       bool *)
  | Fstep b → let x = arg in                                  (* int        int   *)
              apply Fadd (x, if apply Fnot b then 1 else 2)
  | Fbody   → let b = arg in                                  (* bool       int   *)
              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 ∗ 'a expr ∗ 'a expr) → 'a expr
let e0          =  (If (Zerop (Int 0), Int 1, Int 2))
```

What is the type of e0?

Examples — Typed evaluator

A typed abstract-syntax tree

```
type _ expr =
  | Int    : int → int expr
  | Zerop  : int expr → bool expr
  | If     : (bool expr * 'a expr * 'a expr) → 'a 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
```

Examples Typed evaluator

A typed abstract-syntax tree

```
type _ expr =
  | Int    : int → int expr
  | Zerop  : int expr → bool expr
  | If     : (bool expr ∗ 'a expr ∗ 'a expr) → 'a 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

Define a typed abstract syntax tree for the simply-typed lambda-calculus and a *typed* evaluator.

Examples Generic programming

Example of printing

```
type _ ty =
  | Tint : int ty
  | Tbool : bool ty
  | Tlist : 'a ty → ('a list) ty
  | Tpair : 'a ty ∗ 'b ty → ('a ∗ 'b) 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 $*\ \gamma \to (\alpha, \beta)$ 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 $\to (\alpha, \beta)$ 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:

$$\textbf{type}\ (\alpha, \beta)\ \mathsf{eq} = \mathit{Eq} : (\alpha, \alpha)\ \mathsf{eq}$$

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

$$
\begin{array}{lll}
\textbf{type}\ \alpha\ \mathsf{ty} = & & \\
\quad \mid \mathsf{Tint} : (\alpha, \mathsf{int})\ \mathsf{eq} \to \alpha\ \mathsf{ty} & & \color{red}{(*\ \mathit{int\ ty}\ *)} \\
\quad \mid \mathsf{Tlist} : (\alpha, \beta\ \mathsf{list})\ \mathsf{eq} * \beta\ \mathsf{ty} \to \alpha\ \mathsf{ty} & & \color{red}{(*\ \alpha\ \mathit{ty} \to \alpha\ \mathit{list\ ty}\ *)} \\
\quad \mid \mathsf{Tpair} : (\alpha, (\beta * \gamma))\ \mathsf{eq} * \beta\ \mathsf{ty} * \gamma\ \mathsf{ty} \to \alpha\ \mathsf{ty} & &
\end{array}
$$

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 possibkle uses* of the constructors.

Reducing GADTs to type equality (and existential types)

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

$$\textbf{type}\ (\alpha, \beta)\ \mathsf{eq} = \mathit{Eq} : (\alpha, \alpha)\ \mathsf{eq}$$

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!

```
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 :

$$\textbf{type}\ (\alpha, \beta)\ \text{eq} = \textit{Eq} : (\alpha, \alpha)\ \text{eq}$$

For instance, generic programming can be redefined as follows:

$$\begin{array}{l} \textbf{type}\ \alpha\ \text{ty} = \\ \quad \mid \text{Tint} \ : (\alpha, \text{int})\ \text{eq} \to \alpha\ \text{ty} \\ \quad \mid \text{Tlist} \ : (\alpha, \beta\ \text{list})\ \text{eq} * \beta\ \text{ty} \to \alpha\ \text{ty} \\ \quad \mid \text{Tpair} : (\alpha, (\beta * \gamma))\ \text{eq} * \beta\ \text{ty} * \gamma\ \text{ty} \to \alpha\ \text{ty} \end{array}$$

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

$$\begin{array}{l} \textbf{let rec}\ \text{to\_string} : \textbf{type}\ \text{a. a ty} \to \text{a} \to \text{string} = \textbf{fun}\ \text{t x} \to \text{match t with} \\ \mid \text{Tint p} \to \textbf{let}\ \textit{Eq} = \text{p}\ \textbf{in}\ \text{string\_of\_int x} \\ \mid \text{Tlist}\ (\textit{Eq}, \text{l}) \to \ \ldots \\ \mid \text{Tpair}\ (\textit{Eq}, \text{a}, \text{b}) \to \ldots \end{array}$$

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

Formalisation of GADTs

We can encode GADTs with type equalities

We *cannot* encode type equalities in System F.

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
$\mid \gamma_1 \rightarrow \gamma_2$	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:

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

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

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

Fc (simplified) — Typing of coercions

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

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

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

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

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

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

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

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

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

Fc (simplified) — Typing of coercions

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

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

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

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

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

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

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

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

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

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

Semantics

Coercions should be without computational content

?

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

Always?

Semantics

Coercions should be without computational content

Except ...

?

Semantics

Coercions should be without computational content

Except for coercion abstractions that must stop the evaluation

Why?

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,

$\lambda x : \tau_1 \sim \tau_2.\, M$	freezes	the evaluation of M,
$M \triangleleft \gamma$	resumes	the evaluation of M.

  Must always be enforced, even with other strategies
- Full reduction *at compile time*

?

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,

$\lambda x : \tau_1 \sim \tau_2 .\, M$	freezes	the evaluation of M,
$M \triangleleft \gamma$	resumes	the evaluation of M.

  Must always be enforced, even with other strategies
- Full reduction *at compile time* may still be perfomed,

?

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,

$\lambda x : \tau_1 \sim \tau_2.\, M$	freezes	the evaluation of M,
$M \triangleleft \gamma$	resumes	the evaluation of M.

  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.

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

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 $\mathrm{fv}(\lambda x.\, a)$:

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

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

Closure-passing closure conversion

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

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

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 $\mathrm{fv}(\lambda x.\, a)$:

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

Environment-passing closure conversion

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

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

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

Environment-passing closure conversion

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

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

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 $\mathrm{fv}(\lambda x.\, a)$:

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

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

Write $[\![\Gamma]\!]$ for the tuple type $x_1 : [\![\tau_1']\!]; \ldots; x_n : [\![\tau_n']\!]$ where Γ is $x_1 : \tau_1'; \ldots; x_n : \tau_n'$. We also use $[\![\Gamma]\!]$ as a type to mean $[\![\tau_1']\!] \times \ldots \times [\![\tau_n']\!]$.

We have $\Gamma, x : \tau_1 \vdash a : \tau_2$, so in environment $[\![\Gamma]\!], x : [\![\tau_1]\!]$, we have

- env has type $[\![\Gamma]\!]$,
- $code$ has type $([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!]$, and
- the entire closure has type $(([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times [\![\Gamma]\!]$.

Now, *what should be the definition of* $[\![\tau_1 \to \tau_2]\!]$*?*

Towards a type translation

Can we adopt this as a definition?

$$[\![\tau_1 \rightarrow \tau_2]\!] \quad = \quad (([\![\Gamma]\!] \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!]) \times [\![\Gamma]\!]$$

Towards a type translation

Can we adopt this as a definition?

$$[\![\tau_1 \rightarrow \tau_2]\!] \quad = \quad (([\![\Gamma]\!] \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!]) \times [\![\Gamma]\!]$$

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.

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:

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.

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:

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

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 $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\lambda x.\, M)$.

$$
\begin{array}{lcl}
[\![\lambda x{:}\tau_1.\, M]\!] & = & \textit{let } code : \qquad\qquad = \\
& & \quad \lambda(env : \quad, x : \quad). \\
& & \qquad \textit{let } (x_1, \ldots, x_n : \quad) = env \textit{ in} \\
& & \qquad [\![M]\!] \\
& & \textit{in} \\
& & \textbf{pack} \quad , (code, (x_1, \ldots, x_n)) \\
& & \textit{as}
\end{array}
$$

Environment-passing closure conversion

Assume $\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2$ and $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\lambda x.\, M)$.

$$
\begin{array}{rcl}
[\![\lambda x{:}\tau_1.\, M]\!] & = & \textit{let } code : \qquad\qquad\qquad = \\
& & \quad \lambda(env : [\![\Gamma]\!], x : [\![\tau_1]\!]). \\
& & \quad\quad \textit{let } (x_1, \ldots, x_n : [\![\Gamma]\!]) = env \textit{ in} \\
& & \quad\quad [\![M]\!] \\
& & \textit{in} \\
& & \textit{pack} \qquad , (code, (x_1, \ldots, x_n)) \\
& & \textit{as}
\end{array}
$$

Environment-passing closure conversion

Assume $\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2$ and $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\lambda x.\, M)$.

$$
\begin{array}{rcl}
[\![\lambda x{:}\tau_1.\, M]\!] & = & \textit{let } code : ([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
& & \quad \lambda(env : [\![\Gamma]\!], x : [\![\tau_1]\!]). \\
& & \qquad \textit{let } (x_1, \ldots, x_n : [\![\Gamma]\!]) = env \textit{ in} \\
& & \qquad [\![M]\!] \\
& & \textit{in} \\
& & \textit{pack} \qquad , (code, (x_1, \ldots, x_n)) \\
& & \textit{as}
\end{array}
$$

Environment-passing closure conversion

Assume $\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2$ and $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\lambda x.\, M)$.

$$
\begin{array}{rcl}
[\![\lambda x{:}\tau_1.\, M]\!] & = & \textit{let}\ \textit{code} : ([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
& & \quad \lambda(\textit{env} : [\![\Gamma]\!], x : [\![\tau_1]\!]). \\
& & \qquad \textit{let}\ (x_1, \ldots, x_n : [\![\Gamma]\!]) = \textit{env}\ \textit{in} \\
& & \qquad [\![M]\!] \\
& & \textit{in} \\
& & \textit{pack}\ [\![\Gamma]\!], (\textit{code}, (x_1, \ldots, x_n)) \\
& & \textit{as}\ \exists \alpha.((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha
\end{array}
$$

Environment-passing closure conversion

Assume $\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2$ and $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\lambda x.\, M)$.

$$
\begin{array}{rcl}
[\![\lambda x{:}\tau_1.\, M]\!] & = & \textit{let code} : ([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
& & \quad \lambda(env : [\![\Gamma]\!], x : [\![\tau_1]\!]). \\
& & \qquad \textit{let}\ (x_1, \ldots, x_n : [\![\Gamma]\!]) = env\ \textit{in} \\
& & \qquad [\![M]\!] \\
& & \textit{in} \\
& & \textit{pack}\ [\![\Gamma]\!], (code, (x_1, \ldots, x_n)) \\
& & \ \textit{as}\ \exists\alpha.((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha
\end{array}
$$

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 $\Gamma \vdash M_1 : \tau_1$.

$$\begin{aligned}[\![M\ M_1]\!] \quad = \quad & \textit{let}\ \alpha, (code : (\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!], env : \alpha) = \\ & \quad \textit{unpack}\ [\![M]\!]\ \textit{in} \\ & code\ (env, [\![M_1]\!])\end{aligned}$$

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.

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?

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 $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\mu f.\lambda x.M)$.

$$
\begin{aligned}
&[\![\mu f \quad .\lambda x.M]\!] = \\
&\quad \textit{let } code : \quad = \\
&\quad\quad \lambda(env : \quad , x : \quad). \\
&\quad\quad\quad \textit{let } (f, x_1, \ldots, x_n) : \quad = env \textit{ in} \\
&\quad\quad\quad [\![M]\!] \textit{ in} \\
&\quad \textit{let rec } clo : \quad = \\
&\quad\quad \textit{pack} \quad , (code, (clo, x_1, \ldots, x_n)) \\
&\quad\quad \textit{as} \\
&\quad \textit{in } clo
\end{aligned}
$$

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 $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\mu f.\lambda x.M)$.

$$
\begin{array}{l}
[\![\mu f : \tau_1 \to \tau_2.\lambda x.M]\!] = \\
\qquad \textit{let } code : \qquad\qquad\qquad\qquad = \\
\qquad\quad \lambda(env : [\![f : \tau_1 \to \tau_2, \Gamma]\!], x : [\![\tau_1]\!]). \\
\qquad\qquad \textit{let } (f, x_1, \ldots, x_n) : [\![f : \tau_1 \to \tau_2, \Gamma]\!] = env \textit{ in} \\
\qquad\qquad [\![M]\!] \textit{ in} \\
\qquad \textit{let rec } clo : \qquad\qquad = \\
\qquad\quad \textit{pack } [\![f : \tau_1 \to \tau_2, \Gamma]\!], (code, (clo, x_1, \ldots, x_n)) \\
\qquad\quad \textit{as} \\
\qquad \textit{in } clo
\end{array}
$$

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 $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\mu f.\lambda x.M)$.

$$
\begin{array}{l}
[\![\mu f : \tau_1 \to \tau_2.\lambda x.M]\!] = \\
\qquad \textit{let } code : ([\![f : \tau_1 \to \tau_2; \Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
\qquad\quad \lambda(env : [\![f : \tau_1 \to \tau_2, \Gamma]\!], x : [\![\tau_1]\!]). \\
\qquad\qquad \textit{let } (f, x_1, \ldots, x_n) : [\![f : \tau_1 \to \tau_2, \Gamma]\!] = env \textit{ in} \\
\qquad\qquad [\![M]\!] \textit{ in} \\
\qquad \textit{let rec } clo : \qquad\qquad = \\
\qquad\quad \textit{pack } [\![f : \tau_1 \to \tau_2, \Gamma]\!], (code, (clo, x_1, \ldots, x_n)) \\
\qquad\quad \textit{as} \\
\qquad \textit{in } clo
\end{array}
$$

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 $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\mu f.\lambda x.M)$.

$$
\begin{aligned}
&[\![\mu f : \tau_1 \to \tau_2 . \lambda x.M]\!] = \\
&\qquad \textit{let code} : ([\![f : \tau_1 \to \tau_2 ; \Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
&\qquad\quad \lambda(env : [\![f : \tau_1 \to \tau_2, \Gamma]\!], x : [\![\tau_1]\!]). \\
&\qquad\qquad \textit{let } (f, x_1, \ldots, x_n) : [\![f : \tau_1 \to \tau_2, \Gamma]\!] = env \textit{ in} \\
&\qquad\qquad [\![M]\!] \textit{ in} \\
&\qquad \textit{let rec clo} : [\![\tau_1 \to \tau_2]\!] = \\
&\qquad\quad \textit{pack } [\![f : \tau_1 \to \tau_2, \Gamma]\!], (code, (clo, x_1, \ldots, x_n)) \\
&\qquad\quad\ \textit{as} \\
&\qquad \textit{in clo}
\end{aligned}
$$

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 $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\mu f.\lambda x.M)$.

$$
\begin{array}{l}
[\![\mu f : \tau_1 \to \tau_2.\lambda x.M]\!] = \\
\qquad \textit{let } code : ([\![f : \tau_1 \to \tau_2; \Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
\qquad\quad \lambda(env : [\![f : \tau_1 \to \tau_2, \Gamma]\!], x : [\![\tau_1]\!]). \\
\qquad\qquad \textit{let } (f, x_1, \ldots, x_n) : [\![f : \tau_1 \to \tau_2, \Gamma]\!] = env \textit{ in} \\
\qquad\qquad [\![M]\!] \textit{ in} \\
\qquad \textit{let rec } clo : [\![\tau_1 \to \tau_2]\!] = \\
\qquad\quad \textit{pack } [\![f : \tau_1 \to \tau_2, \Gamma]\!], (code, (clo, x_1, \ldots, x_n)) \\
\qquad\quad \textit{as } \exists\alpha((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha) \\
\qquad \textit{in } clo
\end{array}
$$

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 $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\mu f.\lambda x.M)$.

$$
\begin{array}{l}
[\![\mu f : \tau_1 \to \tau_2.\lambda x.M]\!] = \\
\qquad \textit{let code} : ([\![f : \tau_1 \to \tau_2; \Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
\qquad\quad \lambda(env : [\![f : \tau_1 \to \tau_2, \Gamma]\!], x : [\![\tau_1]\!]). \\
\qquad\qquad \textit{let } (f, x_1, \ldots, x_n) : [\![f : \tau_1 \to \tau_2, \Gamma]\!] = env \textit{ in} \\
\qquad\qquad [\![M]\!] \textit{ in} \\
\qquad \textit{let rec clo} : [\![\tau_1 \to \tau_2]\!] = \\
\qquad\quad \textit{pack } [\![f : \tau_1 \to \tau_2, \Gamma]\!], (code, (clo, x_1, \ldots, x_n)) \\
\qquad\quad \textit{as } \exists\alpha((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha) \\
\qquad \textit{in clo}
\end{array}
$$

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

$$
\begin{array}{l}
[\![\mu f{:}\forall\vec{\beta}.\,\tau_1 \to \tau_2.\,\lambda x.M]\!] = \\
\quad \textit{let}\ code : \forall\vec{\beta}.\,([\![f : \forall\vec{\beta}.\,\tau_1 \to \tau_2;\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\
\qquad \lambda(env : [\![f : \forall\vec{\beta}.\,\tau_1 \to \tau_2,\Gamma]\!], x : [\![\tau_1]\!]). \\
\qquad\quad \textit{let}\ (f, x_1, \ldots, x_n) : [\![f : \forall\vec{\beta}.\,\tau_1 \to \tau_2,\Gamma]\!] = env\ \textit{in} \\
\qquad\quad [\![M]\!]\ \textit{in} \\
\quad \textit{let rec}\ clo : [\![\forall\vec{\beta}.\,\tau_1 \to \tau_2]\!] = \\
\qquad \Lambda\vec{\beta}.\,\textit{pack}\ [\![f : \forall\vec{\beta}.\,\tau_1 \to \tau_2,\Gamma]\!], (code\ \vec{\beta}, (clo, x_1, \ldots, x_n)) \\
\qquad\quad \textit{as}\ \exists\alpha((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha) \\
\quad \textit{in}\ clo
\end{array}
$$

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

$$\begin{array}{rcl}
[\![\lambda x.\,M]\!] & = & \textit{let } code = \lambda(clo, x). \\
& & \quad \textit{let } (_, x_1, \ldots, x_n) = clo \textit{ in} \\
& & \quad [\![M]\!] \\
& & \textit{in } (code, x_1, \ldots, x_n) \\
[\![M_1\ M_2]\!] & = & \textit{let } clo = [\![M_1]\!] \textit{ in} \\
& & \textit{let } code = \textit{proj}_0\ clo \textit{ in} \\
& & code\ (clo, [\![M_2]\!])
\end{array}$$

where $\{x_1, \ldots, x_n\} = \mathrm{fv}(\lambda x.\,M)$.

Closure-passing closure conversion

$$
\begin{array}{rcl}
[\![\lambda x.\, M]\!] & = & \textit{let}\ code = \lambda(clo, x). \\
& & \quad \textit{let}\ (\_, x_1, \ldots, x_n) = clo\ \textit{in} \\
& & \quad [\![M]\!] \\
& & \textit{in}\ (code, x_1, \ldots, x_n) \\
\\
[\![M_1\ M_2]\!] & = & \textit{let}\ clo = [\![M_1]\!]\ \textit{in} \\
& & \textit{let}\ code = \textit{proj}_0\ clo\ \textit{in} \\
& & code\ (clo, [\![M_2]\!])
\end{array}
$$

where $\{x_1, \ldots, x_n\} = \mathrm{fv}(\lambda x.\, M)$.

How could we typecheck this? What are the difficulties?

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?

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:

$$\begin{array}{lcll} \tau & ::= & \ldots \mid \Pi\, R & \text{– types} \\ R & ::= & \epsilon \mid (\tau ; R) & \text{– rows} \end{array}$$

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:

$$\begin{array}{lcll} \tau & ::= & \ldots \mid \Pi\, R \mid \forall\, \rho.\, \tau \mid \exists \rho. \tau & \text{– types} \\ R & ::= & \rho \mid \epsilon \mid (\tau ; R) & \text{– rows} \end{array}$$

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:

$$\textsc{Tuple}\;\frac{\forall i.\ \in [1, n] \quad \Gamma \vdash M_i : \tau_i}{\Gamma \vdash (M_1, \ldots, M_n) : \Pi\,(\tau_1; \ldots; \tau_n; \epsilon)}$$

$$\textsc{Proj}\;\frac{\Gamma \vdash M : \Pi\,(\tau_1; \ldots; \tau_i; R)}{\Gamma \vdash \mathit{proj}_i\ M : \tau_i}$$

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:

$$
\begin{array}{lll}
 & [\![\tau_1 \rightarrow \tau_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
 & \quad \mu\alpha. & \alpha \text{ is the concrete type of the closure} \\
 & \quad\quad \Pi\,(& \text{a tuple...} \\
 & \quad\quad\quad (\alpha \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!]; & \text{...that begins with a code pointer...} \\
 & \quad\quad\quad \rho & \text{...and continues with the environment} \\
 & \quad\quad) &
\end{array}
$$

See Morrisett and Harper's “fix-type” encoding [1998].

Closure-passing closure conversion

Rows and recursive types allow to define the translation of types in the closure-passing variant:

$$
\begin{array}{rll}
& [\![\tau_1 \rightarrow \tau_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
& \quad \mu\alpha. & \alpha \text{ is the concrete type of the closure} \\
& \quad\quad \Pi\,(& \text{a tuple...} \\
& \quad\quad\quad (\alpha \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!]; & \text{...that begins with a code pointer...} \\
& \quad\quad\quad \rho & \text{...and continues with the environment} \\
& \quad\quad) &
\end{array}
$$

See Morrisett and Harper's "fix-type" encoding [1998].

Question: Why is it $\exists\rho.\ \mu\alpha.\ \tau$ and not $\mu\alpha.\ \exists\rho.\ \tau$

Closure-passing closure conversion

Rows and recursive types allow to define the translation of types in the closure-passing variant:

$$
\begin{array}{rll}
& [\![\tau_1 \rightarrow \tau_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
& \quad \mu\alpha. & \alpha \text{ is the concrete type of the closure} \\
& \qquad \Pi\,(& \text{a tuple...} \\
& \qquad\quad (\alpha \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!]; & \text{...that begins with a code pointer...} \\
& \qquad\quad \rho & \text{...and continues with the environment} \\
& \qquad) &
\end{array}
$$

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.

Closure-passing closure conversion

Rows and recursive types allow to define the translation of types in the closure-passing variant:

$$
\begin{array}{rll}
 & [\![\tau_1 \rightarrow \tau_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
 & \quad \mu\alpha. & \alpha \text{ is the concrete type of the closure} \\
 & \quad\quad \Pi\,(& \text{a tuple...} \\
 & \quad\quad\quad (\alpha \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!]; & \text{...that begins with a code pointer...} \\
 & \quad\quad\quad \rho & \text{...and continues with the environment} \\
 & \quad\quad) &
\end{array}
$$

See Morrisett and Harper's "fix-type" encoding [1998].

Question: Notice that ρ appears only once. Any comments?

Closure-passing closure conversion

Rows and recursive types allow to define the translation of types in the closure-passing variant:

$$
\begin{array}{lll}
 & [\![\tau_1 \rightarrow \tau_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
 & \quad \mu\alpha. & \alpha \text{ is the concrete type of the closure} \\
 & \quad\quad \Pi\,(& \text{a tuple...} \\
 & \quad\quad\quad (\alpha \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!]; & \text{...that begins with a code pointer...} \\
 & \quad\quad\quad \rho & \text{...and continues with the environment} \\
 & \quad\quad) &
\end{array}
$$

See Morrisett and Harper's "fix-type" encoding [1998].

Question: Notice that ρ appears only once. Any comments?

Usually, an existential type variable appears both at positive and negative occurrences.

Closure-passing closure conversion

Rows and recursive types allow to define the translation of types in the closure-passing variant:

$$
\begin{array}{lll}
 & [\![\tau_1 \to \tau_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
 & \quad \mu\alpha. & \alpha \text{ is the concrete type of the closure} \\
 & \quad\quad \Pi\ (& \text{a tuple...} \\
 & \quad\quad\quad (\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; & \text{...that begins with a code pointer...} \\
 & \quad\quad\quad \rho & \text{...and continues with the environment} \\
 & \quad\quad) &
\end{array}
$$

See Morrisett and Harper's "fix-type" encoding [1998].

Question: Notice that ρ appears only once. Any comments?

Usually, an existential type variable appears both at positive and negative occurrences. Here, the variable appear only at a negative occurrence, but in a recursive part of the type that can be unfolded.

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 $[\![\tau_1 \to \tau_2]\!] = \exists\rho.Clo(\rho)$.

$$
\begin{array}{rcl}
[\![\lambda x : \quad . M]\!] & = & \textit{let } code : \qquad = \\
 & & \quad \lambda(clo : \qquad , x : \quad). \\
 & & \qquad \textit{let } (\_, x_1, \ldots, x_n) : \qquad = \textit{unfold } clo \textit{ in} \\
 & & \qquad [\![M]\!] \textit{ in} \\
 & & \textit{pack} \quad , (\textit{fold } (code, x_1, \ldots, x_n)) \\
 & & \textit{as}
\end{array}
$$

$$
\begin{array}{rcl}
[\![M_1\ M_2]\!] & = & \textit{let } \rho, clo = \textit{unpack } [\![M_1]\!] \textit{ in} \\
 & & \textit{let } code : \qquad = \\
 & & \quad \textit{proj}_0\ (\textit{unfold } clo) \textit{ in} \\
 & & code\ (clo, [\![M_2]\!])
\end{array}
$$

Closure-passing closure conversion

Let $Clo(R)$ abbreviate $\mu\alpha.\Pi\ ((\alpha \times \llbracket\tau_1\rrbracket) \to \llbracket\tau_2\rrbracket; R)$.

Let $UClo(R)$ abbreviate its unfolded version,
$\Pi\ ((Clo(R) \times \llbracket\tau_1\rrbracket) \to \llbracket\tau_2\rrbracket; R)$.

We have $\llbracket\tau_1 \to \tau_2\rrbracket = \exists\rho.Clo(\rho)$.

$$
\begin{array}{rcl}
\llbracket\lambda x{:}\llbracket\tau_1\rrbracket.\,M\rrbracket & = & \textit{let}\ code : (Clo(\llbracket\Gamma\rrbracket) \times \llbracket\tau_1\rrbracket) \to \llbracket\tau_2\rrbracket = \\
 & & \quad \lambda(clo : Clo(\llbracket\Gamma\rrbracket, x : \llbracket\tau_1\rrbracket). \\
 & & \qquad \textit{let}\ (\_, x_1, \ldots, x_n) : UClo\llbracket\Gamma\rrbracket = \textit{unfold}\ clo\ \textit{in} \\
 & & \qquad \llbracket M\rrbracket\ \textit{in} \\
 & & \textit{pack}\ \llbracket\Gamma\rrbracket, (\textit{fold}\ (code, x_1, \ldots, x_n)) \\
 & & \ \textit{as}\ \exists\rho.\,Clo(\rho) \\
 & & \\
\llbracket M_1\ M_2\rrbracket & = & \textit{let}\ \rho, clo = \textit{unpack}\ \llbracket M_1\rrbracket\ \textit{in} \\
 & & \textit{let}\ code : (Clo(\rho) \times \llbracket\tau_1\rrbracket) \to \llbracket\tau_2\rrbracket = \\
 & & \quad \textit{proj}_0\ (\textit{unfold}\ clo)\ \textit{in} \\
 & & code\ (clo, \llbracket M_2\rrbracket)
\end{array}
$$

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:

$$[\![M]\!] \quad =$$

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:

$$\begin{array}{rcl} [\![M]\!] & = & \textit{let } code_i = \lambda(env, x). \\ & & \quad \textit{let } (f_1, f_2, x_1, \ldots, x_n) = env \textit{ in} \\ & & \quad [\![M_i]\!] \\ & & \textit{in} \\ & & \textit{let rec } clo_1 = (code_1, (clo_1, clo_2, x_1, \ldots, x_n)) \\ & & \quad \textit{and } clo_2 = (code_2, (clo_1, clo_2, x_1, \ldots, x_n)) \textit{ in} \\ & & clo_1, clo_2 \end{array}$$

Comments?

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 — Closure passing

Can we compile mutually recursive functions?

$$M \triangleq \mu(f_1, f_2).(\lambda x_1.\, M_1, \lambda x_2.\, M_2)$$

Closure passing:

$$\begin{array}{l}
\textit{let } code_i = \lambda(clo, x). \\
\quad \textit{let } (_\,, f_1, f_2, x_1, \ldots, x_n) = clo \textit{ in } [\![M_i]\!] \\
\textit{in} \\
\textit{let rec } clo_1 = (code_1, clo_1, clo_2, x_1, \ldots, x_n) \\
\quad \textit{and } clo_2 = (code_2, clo_1, clo_2, x_1, \ldots, x_n) \\
\textit{in } clo_1, clo_2
\end{array}$$

Mutually recursive functions — Closure passing

Can we compile mutually recursive functions?

$$M \triangleq \mu(f_1, f_2).(\lambda x_1.\, M_1, \lambda x_2.\, M_2)$$

Closure 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 — Closure passing

Can we compile mutually recursive functions?

$$M \triangleq \mu(f_1, f_2).(\lambda x_1.\, M_1, \lambda x_2.\, M_2)$$

Closure 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?

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:

$$
\begin{array}{l}
class\ c\ (x_1, \ldots x_q) \{ \\
\qquad \textit{meth}\ \ m_1 = M_1 \\
\qquad \ldots \\
\qquad \textit{meth}\ \ m_p = M_p \\
\}
\end{array}
$$

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 $(x_1, \ldots x_q)$.

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 $\exists\rho.\,\mu\alpha.\Pi(\{(m_i : \alpha \to \tau_i)^{i\in I}\}; \rho)$:

$$
\begin{array}{l}
\textit{let}\; m = \{ \\
\quad m_1 = \lambda(m, x_1, \ldots x_q : UClo(R)).\, [\![M_1]\!] \\
\quad \ldots \\
\quad m_p = \lambda(m, x_1, \ldots x_q : UClo(R)).\, [\![M_p]\!] \\
\}\;\; \textit{in} \\
\lambda x_1.\, \ldots \lambda x_q.\, \textit{pack}\; R, \textit{fold}\; (m, x_1, \ldots x_q)\; \textit{as}\; \exists\rho.\,(M, \rho)
\end{array}
$$

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 toplevel polymorphism

- Already, extremely useful! (avoiding dupplication of code)
- 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 $\lambda fxy.(f\ x, f\ y)$

Map on pairs, say *distrib_pair*, has the following types:

Limits of System F $\lambda f x y. (f\ x, f\ y)$

Map on pairs, say *distrib_pair*, has the following types:

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

Limits of System F $\lambda f x y.\,(f\ x, f\ y)$

Map on pairs, say *distrib_pair*, has the following 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.

Limits of System F $\lambda f x y.\,(f\ x, f\ y)$

Map on pairs, say *distrib_pair*, 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.

Limits of System F $\lambda f x y. (f\ x, f\ y)$

Map on pairs, say *distrib_pair*, 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.

Limits of System F $\lambda f x y.\,(f\ x, f\ y)$

Map on pairs, say *distrib_pair*, 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 *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

Introduce kinds κ for types (with a single kind $*$ to stay with System F)

Well-formedness of types becomes $\Gamma \vdash \tau : *$ to check kinds:

$$\frac{\vdash \Gamma \quad \alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : * \quad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall \alpha :: \kappa . \tau : *}$$

$$\vdash \emptyset \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : * \quad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : \tau}$$

Add and check kinds on type abstractions and 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 $\kappa ::= * \mid \kappa \Rightarrow \kappa$

$$\frac{\vdash \Gamma \qquad \alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : * \qquad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall \alpha :: \kappa. \tau : *}$$

New types $\tau ::= \ldots \mid \lambda \alpha :: \kappa. \tau \mid \tau\ \tau$

WfTypeApp

$$\frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_1}$$

WfTypeAbs

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda \alpha :: \kappa_1 . \tau : \kappa_1 \Rightarrow \kappa_2}$$

Typing of expressions is up to type equivalence:

TConv

$$\frac{\Gamma \vdash M : \tau \qquad \tau \equiv_\beta \tau'}{\Gamma \vdash M : \tau'}$$

From System F to System F$^\omega$ Type functions

Redefine kinds as $\kappa ::= * \mid \kappa \Rightarrow \kappa$

$$\frac{\vdash \Gamma \qquad \alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : * \qquad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall \alpha :: \kappa . \tau : *}$$

New types $\tau ::= \ldots \mid \lambda \alpha :: \kappa . \tau \mid \tau\ \tau$

$$\textsc{WfTypeApp}\ \frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_1} \qquad \textsc{WfTypeAbs}\ \frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda \alpha :: \kappa_1 . \tau : \kappa_1 \Rightarrow \kappa_2}$$

Typing of expressions is up to type equivalence:

$$\textsc{TConv}\ \frac{\Gamma \vdash M : \tau \qquad \tau \equiv_\beta \tau'}{\Gamma \vdash M : \tau'}$$

Remark

$$\Gamma \vdash M : \tau \implies \Gamma \vdash \tau : *$$

F^ω, static semantics (altogether on one slide)

Syntax

$$\begin{array}{lcl} \kappa & ::= & * \mid \kappa \Rightarrow \kappa \\ \tau & ::= & \alpha \mid \tau \to \tau \mid \forall \alpha :: \kappa . \tau \mid \lambda \alpha :: \kappa . \tau \mid \tau\ \tau \\ M & ::= & x \mid \lambda x : \tau . M \mid M\ M \mid \Lambda \alpha :: \kappa . M \mid M\ \tau \end{array}$$

Kinding rules

$$\vdash \varnothing \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : * \quad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : \tau} \qquad \frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : * \quad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall \alpha :: \kappa . \tau : *} \qquad \frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda \alpha :: \kappa_1 . \tau : \kappa_1 \Rightarrow \kappa_2} \qquad \frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \quad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_1}$$

Typing rules

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . M : \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\ M_2 : \tau_2}$$

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

F^ω, static semantics (altogether on one slide)

Syntax

$$
\begin{array}{rcl}
\kappa & ::= & * \mid \kappa \Rightarrow \kappa \\
\tau & ::= & \alpha \mid \tau \to \tau \mid \forall \alpha :: \kappa . \tau \mid \lambda \alpha :: \kappa . \tau \mid \tau\ \tau \\
M & ::= & x \mid \lambda x : \tau .\, M \mid M\ M \mid \Lambda \alpha :: \kappa .\, M \mid M\ \tau
\end{array}
$$

Kinding rules

$$\vdash \varnothing \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : * \quad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : \tau} \qquad \frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : * \quad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall \alpha :: \kappa . \tau : *} \qquad \frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda \alpha :: \kappa_1 . \tau : \kappa_1 \Rightarrow \kappa_2} \qquad \frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \quad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_1}$$

Typing rules

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1 .\, M : \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\ M_2 : \tau_2}$$

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

F^ω, static semantics (altogether on one slide)

Syntax

With implicit kinds

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

Kinding rules

$$\vdash \varnothing \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : * \quad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : \tau} \qquad \frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : * \quad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall\alpha.\tau : *} \qquad \frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda\alpha.\tau : \kappa_1 \Rightarrow \kappa_2} \qquad \frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \quad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\;\tau_2 : \kappa_1}$$

Typing rules

$$\text{VAR}\;\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{ABS}\;\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\,M : \tau_1 \to \tau_2} \qquad \text{APP}\;\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\;M_2 : \tau_2}$$

$$\text{TABS}\;\frac{\Gamma, \alpha : \kappa \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\,M : \forall\alpha.\tau} \qquad \text{TAPP}\;\frac{\Gamma \vdash M : \forall\alpha.\tau \quad \Gamma \vdash \tau' : \kappa}{\Gamma \vdash M\;\tau' : [\alpha \mapsto \tau']\tau} \qquad \text{TEQUIV}\;\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash \tau \equiv_\beta \tau'}{\Gamma \vdash M : \tau'}$$

F^ω, dynamic semantics

The semantics is unchanged (modulo kind annotations in terms)

$$
\begin{array}{lcl}
V & ::= & \lambda x\!:\!\tau.\, M \mid \Lambda\alpha :: \kappa.\, V \\
E & ::= & [\,]\, M \mid V\, [\,] \mid [\,]\, \tau \mid \Lambda\alpha :: \kappa.\, [\,]
\end{array}
$$

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

$$
(\Lambda\alpha :: \kappa.\, V)\, \tau \longrightarrow [\alpha \mapsto \tau]V
$$

Context

$$
\frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}
$$

No type reduction

- We need not reduce types inside terms.
- Type reduction is needed for type conversion (*i.e.* for typing) but such reduction need not be performed on terms.

Kinds are erasable

- Reduction preserves kinds.
- Kinds are just ignored during the reduction (they need not be reduced). In fact, kinds can be 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, but for *full reduction* (we have only proved it for CBV).

It is a key that reduction terminates.
(Again, we have only proved it for CBV.)

Contents

Expressiveness

More polymorphism

- distrib_pair

Abstraction over type operators

- monads
- encoding of existentials

Encodings

- non regular datatypes
- equality

Distrib pair in F^ω $\lambda f x y.\,(f\ x, f\ y)$

Abstract over (one parameter) type *functions* (*e.g.* of kind $\star \to \star$)

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

call it distrib_pair of type:

$$\forall\varphi :: \ast \Rightarrow \ast.\forall\psi :: \ast \Rightarrow \ast.\forall\alpha_1 :: \ast.\forall\alpha_2 :: \ast.$$
$$(\forall\alpha :: \ast.\,\varphi\alpha \to \psi\alpha) \to \varphi\alpha_1 \to \varphi\alpha_2 \to \psi\alpha_1 \times \psi\alpha_2$$

We may recover, in particular, the two types it had in System F:

$$\Lambda\alpha_1 :: \ast.\,\Lambda\alpha_2 :: \ast.\,\text{distrib\_pair}\ (\lambda\alpha :: \ast.\,\alpha_1)\ (\lambda\alpha :: \ast.\,\alpha_2)\ \alpha_1\ \alpha_2$$
$$: \forall\alpha_1 :: \ast.\,\forall\alpha_2 :: \ast.\,(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

$$\text{distrib\_pair}\ (\lambda\alpha :: \ast.\alpha)\ (\lambda\alpha :: \ast.\alpha)$$
$$: \forall\alpha_1 :: \ast.\,\forall\alpha_2.\,(\forall\alpha :: \ast.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

Distrib pair in F^ω (with implicit kinds) $\lambda fxy.\,(f\ x, f\ y)$

Abstract over (one parameter) type *functions* (*e.g.* of kind $\star \to \star$)

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

call it distrib_pair 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 had in System F:

$$\Lambda\alpha_1.\,\Lambda\alpha_2.\,\text{distrib\_pair}\ (\lambda\alpha.\,\alpha_1)\ (\lambda\alpha.\,\alpha_2)\ \alpha_1\ \alpha_2$$
$$: \forall\alpha_1.\,\forall\alpha_2.\,(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

$$\text{distrib\_pair}\ (\lambda\alpha.\alpha)\ (\lambda\alpha.\alpha)$$
$$: \forall\alpha_1.\,\forall\alpha_2.\,(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

Distrib pair in F^ω (with implicit kinds) $\lambda fxy.\,(f\ x, f\ y)$

Abstract over (one parameter) type *functions* (*e.g.* of kind $\star \to \star$)

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

call it distrib_pair 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 had in System F:

$$\Lambda\alpha_1.\,\Lambda\alpha_2.\,\text{distrib\_pair}\ (\lambda\alpha.\,\alpha_1)\ (\lambda\alpha.\,\alpha_2)\ \alpha_1\ \alpha_2$$
$$: \forall\alpha_1.\,\forall\alpha_2.\,(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

$$\text{distrib\_pair}\ (\lambda\alpha.\alpha)\ (\lambda\alpha.\alpha)$$
$$: \forall\alpha_1.\,\forall\alpha_2.\,(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

Still, the type of distrib_pair is not principal. φ and ψ could depend on two variables, *i.e.* be of kind $* \Rightarrow * \Rightarrow *$, 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).

$$
\begin{aligned}
M \;&\triangleq\; \lambda\,(\varphi :: * \Rightarrow *)\,.\\
&\qquad \{\ \mathit{ret} : \forall\,(\alpha :: *)\,.\,\alpha \to \varphi\alpha;\\
&\qquad\quad \mathit{bind} : \forall\,(\alpha :: *)\,.\,\forall\,(\beta :: *)\,.\,\varphi\alpha \to (\alpha \to \varphi\beta) \to \varphi\beta\ \}\\
&:\; (* \Rightarrow *) \Rightarrow *
\end{aligned}
$$

(Notice that M is itself of higher kind)

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).

$$\begin{array}{rcl} M & \triangleq & \lambda\,(\varphi :: * \Rightarrow *)\,. \\ & & \quad \{\ \mathit{ret} : \forall\,(\alpha :: *)\,.\,\alpha \to \varphi\alpha; \\ & & \quad\ \ \mathit{bind} : \forall\,(\alpha :: *)\,.\,\forall\,(\beta :: *)\,.\,\varphi\alpha \to (\alpha \to \varphi\beta) \to \varphi\beta\ \} \\ & : & (* \Rightarrow *) \Rightarrow * \end{array}$$

(Notice that M is itself of higher kind)

A generic map function: can then be defined:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{fmap}} \\ \triangleq & \Lambda\,(\varphi :: * \Rightarrow *)\,.\,\lambda m : M\,\varphi. \\ & \qquad \Lambda\,(\alpha :: *)\,.\,\Lambda\,(\beta :: *)\,.\,\lambda f : (\alpha \to \beta).\,\lambda x : \varphi\alpha. \\ & \qquad\qquad m.\mathit{bind}\;\alpha\;\beta\;x\;(\lambda x : \alpha.\,m.\mathit{ret}\;(f\;x)) \\ : & \forall\,(\varphi :: * \Rightarrow *)\,.\,M\,\varphi \to \forall\,(\alpha :: *)\,.\,\forall\,(\beta :: *)\,.\,(\alpha \to \beta) \to \varphi\alpha \to \varphi\beta \end{array}$$

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).

$$
\begin{aligned}
M \;\triangleq\;& \lambda\varphi.\\
&\quad \{\; \mathit{ret} : \forall\alpha.\,\alpha \to \varphi\alpha;\\
&\quad\;\; \mathit{bind} : \forall\alpha.\,\forall\beta.\,\varphi\alpha \to (\alpha \to \varphi\beta) \to \varphi\beta \;\}\\
:\;& (* \Rightarrow *) \Rightarrow *
\end{aligned}
$$

(Notice that M is itself of higher kind)

A generic map function: can then be defined:

$$
\begin{aligned}
&\mathit{fmap}\\
&\quad\triangleq\quad \Lambda\varphi.\,\lambda m : M\,\varphi.\\
&\qquad\qquad \Lambda\alpha.\,\Lambda\beta.\,\lambda f : (\alpha \to \beta).\,\lambda x : \varphi\alpha.\\
&\qquad\qquad\qquad m.\mathit{bind}\;\alpha\;\beta\;x\;(\lambda x : \alpha.\,m.\mathit{ret}\;(f\;x))\\
&\quad:\quad \forall\varphi.\,M\,\varphi \to \forall\alpha.\,\forall\beta.\,(\alpha \to \beta) \to \varphi\alpha \to \varphi\beta
\end{aligned}
$$

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).

$$
\begin{aligned}
M \;&\triangleq\; \lambda\varphi.\\
&\qquad \{\, \mathit{ret} : \forall\alpha.\, \alpha \to \varphi\alpha;\\
&\qquad\;\; \mathit{bind} : \forall\alpha.\, \forall\beta.\, \varphi\alpha \to (\alpha \to \varphi\beta) \to \varphi\beta \,\}\\
&:\; (\ast \Rightarrow \ast) \Rightarrow \ast
\end{aligned}
$$

(Notice that M is itself of higher kind)

A generic map function: can then be defined:

$$
\begin{aligned}
&\mathit{fmap}\\
&\quad\triangleq\quad \lambda m.\\
&\qquad\qquad \lambda f.\, \lambda x.\\
&\qquad\qquad\qquad m.\mathit{bind}\, x\, (\lambda x.\, m.\mathit{ret}\, (f\ x))\\
&\quad:\quad \forall\varphi.\, M\,\varphi \to \forall\alpha.\, \forall\beta.\, (\alpha \to \beta) \to \varphi\alpha \to \varphi\beta
\end{aligned}
$$

Abstracting over type operators

Available in Haskell —without β-reduction

- $\varphi\alpha$ is treated as a type $App(\varphi, \alpha)$ where
  $App : (\kappa_1 \Rightarrow \kappa_2) \Rightarrow \kappa_1 \Rightarrow \kappa_2$
- 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 does enable 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.

Encoding of existentials

Limits of System F

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \quad = \quad ?$$

Encoding of existentials

Limits of System F

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Encoding of existentials — Limits of System F

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;=\; \Lambda\alpha.\,\lambda x{:}[\![\tau]\!].\,\Lambda\beta.\,\lambda k{:}\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

Encoding of existentials — Limits of System F

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;=\; \Lambda\alpha.\,\lambda x\!:\![\![\tau]\!].\,\Lambda\beta.\,\lambda k\!:\!\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we need to abstract over the type function $\lambda\alpha.\,\tau$:

Encoding of existentials in System F^ω

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;=\; \Lambda\alpha.\,\lambda x\!:\![\![\tau]\!].\,\Lambda\beta.\,\lambda k\!:\!\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we need to abstract over the type function $\lambda\alpha.\,\tau$:

In System F^ω, we may defined

$$[\![\mathit{pack}\;\;]\!] \;=\; ?$$

Encoding of existentials in System F^ω

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;=\; \Lambda\alpha.\,\lambda x\!:\![\![\tau]\!].\,\Lambda\beta.\,\lambda k\!:\!\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we need to abstract over the type function $\lambda\alpha.\,\tau$:

In System F^ω, we may defined

$$[\![\mathit{pack}\;\;]\!] \;=\; \Lambda\,(\varphi :: \star \Rightarrow \star)\,.\,?$$

Encoding of existentials in System F^ω

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;=\; \Lambda\alpha.\,\lambda x\!:\![\![\tau]\!].\,\Lambda\beta.\,\lambda k\!:\!\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we need to abstract over the type function $\lambda\alpha.\,\tau$:

In System F^ω, we may defined

$$\begin{aligned}[\![\mathit{pack}\;]\!] \;=\; & \Lambda\,(\varphi :: * \Rightarrow *)\,.\,\Lambda\,(\alpha :: *)\,.\\ & \lambda x : \varphi\;\alpha.\,\Lambda\,(\beta :: *)\,.\,\lambda k : \forall\,(\alpha :: *)\,.\,(\varphi\;\alpha \to \beta).\,k\;\alpha\;x\end{aligned}$$

Encoding of existentials in System F^ω

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\textit{pack}_{\exists\alpha.\tau}]\!] \;=\; \Lambda\alpha.\,\lambda x\!:\![\![\tau]\!].\,\Lambda\beta.\,\lambda k\!:\!\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we need to abstract over the type function $\lambda\alpha.\,\tau$:

In System F^ω, we may defined

$$[\![\textit{pack}\;\;]\!] \;=\; \Lambda\varphi.\,\Lambda\alpha.\,\lambda x : \varphi\;\alpha.\,\Lambda\beta.\,\lambda k : \forall\alpha.\,(\varphi\;\alpha \to \beta).\,k\;\alpha\;x$$

(omitting kinds)

Encoding of existentials in System F^ω

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;=\; \Lambda\alpha.\,\lambda x\!:\![\![\tau]\!].\,\Lambda\beta.\,\lambda k\!:\!\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we need to abstract over the type function $\lambda\alpha.\,\tau$:

In System F^ω, we may defined

$$\begin{aligned}[\![\mathit{pack}_{\kappa}]\!] \;=\; & \Lambda\,(\varphi :: \kappa \Rightarrow *)\,.\,\Lambda\,(\alpha :: \kappa)\,.\\ & \lambda x : \varphi\;\alpha.\,\Lambda\,(\beta :: *)\,.\,\lambda k : \forall\,(\alpha :: \kappa)\,.\,(\varphi\;\alpha \to \beta).\,k\;\alpha\;x\end{aligned}$$

Allows abstraction at higher kinds!

Exploiting kinds

Once we have kind functions, the language of types could be reduced to λ-calculus with constants (plus the arrow types kept as primitive):

$$\tau = \alpha \mid \lambda\alpha.\,\tau \mid \tau\ \tau \mid \tau \to \tau \mid g$$

where type constants $g \in \mathcal{G}$ are given with their kind and syntactic sugar:

$$\begin{array}{rcl} \times & :: & * \Rightarrow * \Rightarrow * \\ + & :: & * \Rightarrow * \Rightarrow \kappa \\ \forall & :: & (\kappa \Rightarrow *) \Rightarrow * \\ \exists & :: & (\kappa \Rightarrow *) \Rightarrow * \end{array} \qquad \begin{array}{lcl} (\tau \times \tau) & \triangleq & (\times)\ \tau_1\ \tau_2 \\ (\tau + \tau) & \triangleq & (+)\ \tau_1\ \tau_2 \\ \forall\varphi.\,\tau & \triangleq & \forall(\lambda\varphi.\,\tau) \\ \exists\varphi.\,\tau & \triangleq & \exists(\lambda\varphi.\,\tau) \end{array}$$

Church encoding of regular ADT — List

$$\begin{array}{ll} \texttt{type} & \mathit{List}\ \alpha = \\ & \mid \mathit{Nil} \quad : \forall \alpha.\ \mathit{List}\ \alpha \\ & \mid \mathit{Cons} : \forall \alpha.\ \alpha \to \mathit{List}\ \alpha \to \mathit{List}\ \alpha \end{array}$$

Church encoding (CPS style) in System F

$$\begin{array}{lcl} \mathit{List} & \triangleq & \boxed{\lambda\alpha.\ \forall\beta.\ \beta \to (\alpha \to \beta \to \beta) \to \beta} \\ \mathit{Nil} & \triangleq & \lambda n.\ \lambda c.\ n \\ \mathit{Cons} & \triangleq & \lambda x.\ \lambda \ell.\ \lambda n.\ \lambda c.\ c\ x\ (\ell\ \beta\ n\ c) z \end{array}$$

$$\mathit{fold} \triangleq \lambda n.\ \lambda c.\ \lambda \ell.\ \ell\ \beta\ n\ c$$

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

$$
\begin{array}{lcl}
\mathit{List} & \triangleq & \boxed{\lambda\alpha.\,\forall\beta.\,\beta \to (\alpha \to \beta \to \beta) \to \beta} \\
\mathit{Nil} & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\lambda n : \beta.\,\lambda c : (\alpha \to \beta \to \beta).\,n \\
\mathit{Cons} & \triangleq & \Lambda\alpha.\,\lambda x : \alpha.\,\lambda \ell : \mathit{List}\,\alpha. \\
& & \quad \Lambda\beta.\,\lambda n : \beta.\,\lambda c : (\alpha \to \beta \to \beta).\,c\;x\;(\ell\;\beta\;n\;c)z \\
\mathit{fold} & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\lambda n : \beta.\,\lambda c : (\alpha \to \beta \to \beta).\,\lambda \ell : \mathit{List}\,\alpha.\,\ell\;\beta\;n\;c
\end{array}
$$

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) enhanced in F^ω ?

$$\mathit{List} \triangleq \boxed{\lambda\alpha.\, \forall\varphi.\, \varphi\alpha \to (\alpha \to \varphi\alpha \to \varphi\alpha) \to \varphi\alpha}$$

$$\mathit{Nil} \triangleq \Lambda\alpha.\, \Lambda\varphi.\, \lambda n : \varphi\alpha.\, \lambda c : (\alpha \to \varphi\alpha \to \varphi\alpha).\, n$$

$$\mathit{Cons} \triangleq \Lambda\alpha.\, \lambda x : \alpha.\, \lambda\ell : \mathit{List}\ \alpha.\ \Lambda\varphi.\, \lambda n : \varphi\alpha.\, \lambda c : (\alpha \to \varphi\alpha \to \varphi\alpha).\, c\ x\ (\ell\ \varphi\ n\ c)z$$

$$\mathit{fold} \triangleq \Lambda\alpha.\, \Lambda\varphi.\, \lambda n : \varphi\alpha.\, \lambda c : (\alpha \to \varphi\alpha \to \varphi\alpha).\, \lambda\ell : \mathit{List}\ \alpha.\, \ell\ \varphi\ n\ c$$

Actually not !

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

$$\begin{array}{ll} \texttt{type} & \mathit{Seq}\ \alpha = \\ \quad \mid \mathit{Nil} & : \forall\alpha.\, \mathit{Seq}\ \alpha \\ \quad \mid \mathit{Zero} & : \forall\alpha.\, \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha \\ \quad \mid \mathit{One} & : \forall\alpha.\, \alpha \to \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha \end{array}$$

Encoded as:

$$\mathit{Seq} \triangleq \lambda\alpha.\, \forall F.\, F\alpha \to (F(\alpha\times\alpha) \to F\alpha) \to (\alpha \to F(\alpha\times\alpha) \to F\alpha) \to F\alpha$$

$$\begin{aligned} \mathit{Nil} &\triangleq \lambda n.\, \lambda z.\, \lambda s.\, n \\ \mathit{Zero} &\triangleq \lambda \ell.\, \lambda n.\, \lambda z.\, \lambda s.\, z\ (\ell\ n\ z\ s) \\ \mathit{One} &\triangleq \lambda x.\, \lambda \ell.\, \lambda n.\, \lambda z.\, \lambda s.\, s\ x\ (\ell\ n\ z\ s) \end{aligned}$$

$$\mathit{fold} \triangleq \lambda n.\, \lambda z.\, \lambda s.\, \lambda \ell.\, \ell\ n\ z\ s$$

Church encoding of *non*-regular ADTs — Okasaki's Seq

$$\begin{array}{ll}
\texttt{type} & \mathit{Seq}\ \alpha = \\
\quad \mid \mathit{Nil} & : \forall\alpha.\, \mathit{Seq}\ \alpha \\
\quad \mid \mathit{Zero} & : \forall\alpha.\, \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha \\
\quad \mid \mathit{One} & : \forall\alpha.\, \alpha \to \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha
\end{array}$$

Encoded as:

$$\mathit{Seq} \triangleq \lambda\alpha.\, \forall F.\, F\alpha \to (F(\alpha\times\alpha) \to F\alpha) \to (\alpha \to F(\alpha\times\alpha) \to F\alpha) \to F\alpha$$

$$\mathit{Nil} \triangleq \Lambda\alpha.\, \Lambda F.\, \lambda n : F\alpha.\, \lambda z : F(\alpha\times\alpha) \to F\alpha.\, \lambda s : \alpha \to F(\alpha\times\alpha) \to F\alpha.\, n$$

$$\mathit{Zero} \triangleq \Lambda\alpha.\, \lambda\ell : \mathit{Seq}\ \alpha.\ \cdots$$

$$\begin{aligned}
\mathit{One} \triangleq\ & \Lambda\alpha.\, \lambda x : \alpha.\, \lambda\ell : \mathit{Seq}\ \alpha. \\
& \quad \Lambda F.\, \lambda n : F\alpha.\, \lambda z : F(\alpha\times\alpha) \to F\alpha.\, \lambda s : \alpha \to F(\alpha\times\alpha) \to F\alpha. \\
& \qquad s\ x\ (\ell\ F\ n\ z\ s)
\end{aligned}$$

$$\begin{aligned}
\mathit{fold} \triangleq\ & \Lambda\alpha.\, \Lambda F.\, \lambda n : F\alpha.\, \lambda z : F(\alpha\times\alpha) \to F\alpha.\, \lambda s : \alpha \to F(\alpha\times\alpha) \to F\alpha. \\
& \qquad \lambda\ell : \mathit{Seq}\ \alpha.\, \ell\ F\ n\ z\ s
\end{aligned}$$

Church encoding of *non*-regular ADTs Okasaki's Seq

$$
\begin{array}{ll}
\texttt{type} & \mathit{Seq}\ \alpha = \\
\quad \mid \mathit{Nil} & : \forall\alpha.\ \mathit{Seq}\ \alpha \\
\quad \mid \mathit{Zero} & : \forall\alpha.\ \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha \\
\quad \mid \mathit{One} & : \forall\alpha.\ \alpha \to \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha
\end{array}
$$

Encoded as:

$$\mathit{Seq} \triangleq \boxed{\lambda\alpha.\ \forall F.\ F\alpha \to (F(\alpha\times\alpha) \to F\alpha) \to (\alpha \to F(\alpha\times\alpha) \to F\alpha) \to F\alpha}$$

$$\mathit{Nil} \triangleq \Lambda\alpha.\ \Lambda F.\ \lambda n : F\alpha.\ \lambda z : F(\alpha\times\alpha) \to F\alpha.\ \lambda s : \alpha \to F(\alpha\times\alpha) \to F\alpha.\ n$$

$$\mathit{Zero} \triangleq \Lambda\alpha.\ \lambda\ell : \mathit{Seq}\ \alpha.\ \ \cdots$$

$$\mathit{One} \triangleq \Lambda\alpha.\ \lambda x : \alpha.\ \lambda\ell : \mathit{Seq}\ \alpha.\ \Lambda F.\ \lambda n : F\alpha.\ \lambda z : F(\alpha\times\alpha) \to F\alpha.\ \lambda s : \alpha \to F(\alpha\times\alpha) \to F\alpha.\ s\ x\ (\ell\ F\ n\ z\ s)$$

$$\mathit{fold} \triangleq \Lambda\alpha.\ \Lambda F.\ \lambda n : F\alpha.\ \lambda z : F(\alpha\times\alpha) \to F\alpha.\ \lambda s : \alpha \to F(\alpha\times\alpha) \to F\alpha.\ \lambda\ell : \mathit{Seq}\ \alpha.\ \ell\ F\ n\ z\ s$$

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 ('a, 'b) eq = Eq : ('a, 'a) eq

  let coerce (type a) (type b) (ab : (a,b) eq) (x : a) : b = let Eq = ab in x

  let refl : ('a, 'a) eq = Eq

  (* all these are propagation are 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

$$\mathit{Eq}\ \alpha\ \beta \equiv \forall\varphi.\, \varphi\alpha \to \varphi\beta$$

$$
\begin{array}{lcl}
\mathit{Eq} & \triangleq & \lambda\alpha.\, \lambda\beta.\, \forall\varphi.\, \varphi\alpha \to \varphi\beta \\
\mathit{coerce} & \triangleq & \Lambda\alpha.\, \Lambda\beta.\, \lambda p : \mathit{Eq}\ \alpha\ \beta.\, \lambda x : \alpha.\, p\ (\lambda\gamma.\, \gamma)\ x \\
\mathit{refl} & \triangleq & \Lambda\alpha.\, \Lambda\varphi.\, \lambda x : \varphi\alpha.\ x \\
 & : & \forall\alpha.\, \forall\varphi.\, \varphi\alpha \to \varphi\alpha \quad \equiv \quad \forall\alpha.\, \mathit{Eq}\ \alpha\ \alpha \\
\mathit{symm} & \triangleq & \Lambda\alpha.\, \Lambda\beta.\, \lambda p : \mathit{Eq}\ \alpha\ \beta.\ p\ (\lambda\gamma.\, \mathit{Eq}\ \gamma\ \alpha)\ (\mathit{refl}\ \alpha) \\
 & : & \forall\alpha.\, \forall\beta.\, \mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \alpha \\
\mathit{trans} & \triangleq & \Lambda\alpha.\, \Lambda\beta.\, \Lambda\gamma.\, \lambda p : \mathit{Eq}\ \alpha\ \beta.\, \lambda q : \mathit{Eq}\ \beta\ \gamma.\ q\ (\mathit{Eq}\ \alpha)\ p \\
 & : & \forall\alpha.\, \forall\beta.\, \forall\gamma.\, \mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \gamma \to \mathit{Eq}\ \alpha\ \gamma \\
\mathit{lift} & \triangleq & \Lambda\alpha.\, \Lambda\beta.\, \Lambda\varphi.\, \lambda p : \mathit{Eq}\ \alpha\ \beta.\ p\ (\lambda\gamma.\, \mathit{Eq}\ (\varphi\alpha)\ (\varphi\gamma))\ (\mathit{refl}\ (\varphi\alpha)) \\
 & : & \forall\alpha.\, \forall\beta.\, \forall\varphi.\, \mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ (\varphi\alpha)\ (\varphi\beta)
\end{array}
$$

Equality

Leibnitz equality in F^ω

$$\mathit{Eq}\ \alpha\ \beta \equiv \forall \varphi.\, \varphi\alpha \to \varphi\beta$$

$$
\begin{array}{lcl}
\mathit{Eq} & \triangleq & \lambda\alpha.\, \lambda\beta.\, \forall\varphi.\, \varphi\alpha \to \varphi\beta \\
\mathit{coerce} & \triangleq & \Lambda\alpha.\, \Lambda\beta.\, \lambda p : \mathit{Eq}\ \alpha\ \beta.\, \lambda x : \alpha.\, p\ (\lambda\gamma.\, \gamma)\ x \\
\mathit{refl} & \triangleq & \Lambda\alpha.\, \Lambda\varphi.\, \lambda x : \varphi\alpha.\ x \\
 & : & \forall\alpha.\, \forall\varphi.\, \varphi\alpha \to \varphi\alpha \quad \equiv \quad \forall\alpha.\, \mathit{Eq}\ \alpha\ \alpha \\
\mathit{symm} & \triangleq & \Lambda\alpha.\, \Lambda\beta.\, \lambda p : \mathit{Eq}\ \alpha\ \beta.\ p\ (\lambda\gamma.\, \mathit{Eq}\ \gamma\ \alpha)\ (\mathit{refl}\ \alpha) \\
 & : & \forall\alpha.\, \forall\beta.\, \mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \alpha \qquad : \mathit{Eq}\ \alpha\ \alpha \to \mathit{Eq}\ \beta\ \alpha \\
\mathit{trans} & \triangleq & \Lambda\alpha.\, \Lambda\beta.\, \Lambda\gamma.\, \lambda p : \mathit{Eq}\ \alpha\ \beta.\, \lambda q : \mathit{Eq}\ \beta\ \gamma.\ q\ (\mathit{Eq}\ \alpha)\ p \\
 & : & \forall\alpha.\, \forall\beta.\, \forall\gamma.\, \mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \gamma \to \mathit{Eq}\ \alpha\ \gamma \\
\mathit{lift} & \triangleq & \Lambda\alpha.\, \Lambda\beta.\, \Lambda\varphi.\, \lambda p : \mathit{Eq}\ \alpha\ \beta.\ p\ (\lambda\gamma.\, \mathit{Eq}\ (\varphi\alpha)\ (\varphi\gamma))\ (\mathit{refl}\ (\varphi\alpha)) \\
 & : & \forall\alpha.\, \forall\beta.\, \forall\varphi.\, \mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ (\varphi\alpha)\ (\varphi\beta)
\end{array}
$$

Equality

Leibnitz equality in F^ω

$\boxed{\textit{Eq}\ \alpha\ \beta \equiv \forall \varphi.\, \varphi\alpha \to \varphi\beta}$

$$
\begin{array}{lcl}
\textit{Eq} & \triangleq & \lambda\alpha.\,\lambda\beta.\,\forall\varphi.\,\varphi\alpha \to \varphi\beta \\
\textit{coerce} & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\lambda p : \textit{Eq}\ \alpha\ \beta.\,\lambda x : \alpha.\, p\ (\lambda\gamma.\,\gamma)\ x \\
\textit{refl} & \triangleq & \Lambda\alpha.\,\Lambda\varphi.\,\lambda x : \varphi\alpha.\ x \\
 & : & \forall\alpha.\,\forall\varphi.\,\varphi\alpha \to \varphi\alpha \ \equiv\ \forall\alpha.\,\textit{Eq}\ \alpha\ \alpha \\
\textit{symm} & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\lambda p : \textit{Eq}\ \alpha\ \beta.\ p\ (\lambda\gamma.\,\textit{Eq}\ \gamma\ \alpha)\ (\textit{refl}\ \alpha) \\
 & : & \forall\alpha.\,\forall\beta.\,\textit{Eq}\ \alpha\ \beta \to \textit{Eq}\ \beta\ \alpha \\
\textit{trans} & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\Lambda\gamma.\,\lambda p : \textit{Eq}\ \alpha\ \beta.\,\lambda q : \textit{Eq}\ \beta\ \gamma.\ \boxed{q\ (\textit{Eq}\ \alpha)}\ p \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\gamma.\,\textit{Eq}\ \alpha\ \beta \to \textit{Eq}\ \beta\ \gamma \to \textit{Eq}\ \alpha\ \gamma \quad \boxed{: \textit{Eq}\ \alpha\ \beta \to \textit{Eq}\ \alpha\ \gamma} \\
\textit{lift} & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\Lambda\varphi.\,\lambda p : \textit{Eq}\ \alpha\ \beta.\ p\ (\lambda\gamma.\,\textit{Eq}\ (\varphi\alpha)\ (\varphi\gamma))\ (\textit{refl}\ (\varphi\alpha)) \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\varphi.\,\textit{Eq}\ \alpha\ \beta \to \textit{Eq}\ (\varphi\alpha)\ (\varphi\beta)
\end{array}
$$

Equality

Leibnitz equality in F^ω

$$Eq\ \alpha\ \beta \equiv \forall \varphi.\, \varphi\alpha \to \varphi\beta$$

$$\begin{array}{lcl}
Eq & \triangleq & \lambda\alpha.\,\lambda\beta.\,\forall\varphi.\,\varphi\alpha \to \varphi\beta \\
coerce & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\lambda p : Eq\ \alpha\ \beta.\,\lambda x : \alpha.\, p\ (\lambda\gamma.\,\gamma)\ x \\
refl & \triangleq & \Lambda\alpha.\,\Lambda\varphi.\,\lambda x : \varphi\alpha.\ x \\
 & : & \forall\alpha.\,\forall\varphi.\,\varphi\alpha \to \varphi\alpha \quad \equiv \quad \forall\alpha.\, Eq\ \alpha\ \alpha \\
symm & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\lambda p : Eq\ \alpha\ \beta.\ p\ (\lambda\gamma.\, Eq\ \gamma\ \alpha)\ (refl\ \alpha) \\
 & : & \forall\alpha.\,\forall\beta.\, Eq\ \alpha\ \beta \to Eq\ \beta\ \alpha \\
trans & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\Lambda\gamma.\,\lambda p : Eq\ \alpha\ \beta.\,\lambda q : Eq\ \beta\ \gamma.\ q\ (Eq\ \alpha)\ p \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\gamma.\, Eq\ \alpha\ \beta \to Eq\ \beta\ \gamma \to Eq\ \alpha\ \gamma \\
lift & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\Lambda\varphi.\,\lambda p : Eq\ \alpha\ \beta.\ p\ (\lambda\gamma.\, Eq\ (\varphi\alpha)\ (\varphi\gamma))\ (refl\ (\varphi\alpha)) \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\varphi.\, Eq\ \alpha\ \beta \to Eq\ (\varphi\alpha)\ (\varphi\beta)
\end{array}$$

Equality

Leibnitz equality in F^ω

$\mathit{Eq}\ \alpha\ \beta \equiv \forall \varphi.\, \varphi\alpha \to \varphi\beta$

$$
\begin{array}{lcl}
\mathit{Eq} & \triangleq & \lambda\alpha.\,\lambda\beta.\,\forall\varphi.\,\varphi\alpha \to \varphi\beta \\
\mathit{coerce} & \triangleq & \lambda p.\,\lambda x.\,p\ x \\
\mathit{refl} & \triangleq & \lambda x.\ x \\
 & : & \forall\alpha.\,\forall\varphi.\,\varphi\alpha \to \varphi\alpha \quad\equiv\quad \forall\alpha.\,\mathit{Eq}\ \alpha\ \alpha \\
\mathit{symm} & \triangleq & \lambda p.\ p\ (\mathit{refl}) \\
 & : & \forall\alpha.\,\forall\beta.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \alpha \\
\mathit{trans} & \triangleq & \lambda p.\,\lambda q.\ q\ p \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\gamma.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \gamma \to \mathit{Eq}\ \alpha\ \gamma \\
\mathit{lift} & \triangleq & \lambda p.\ p\ (\mathit{refl}) \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\varphi.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ (\varphi\alpha)\ (\varphi\beta)
\end{array}
$$

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 the type operator, 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.

A hierarchy of type systems

Kinds have a rank:

- the base kind $*$ is of rank 0
- kinds $* \Rightarrow *$ and $* \Rightarrow * \Rightarrow *$ have rank 1. They are the kinds of type functions taking type parameters of base kind.
- kind $(* \Rightarrow *) \Rightarrow *$ has rank 2—it is a type function whose parameter is itself a simple type function (of rank 1).
- more generally, $\mathit{rank}\,(\kappa_1 \Rightarrow \kappa_2) = \max(1 + \mathit{rank}\,\kappa_1, \mathit{rank}\,\kappa_2)$

This defines a sequence $F^0 \subseteq F^1 \subseteq F^2 \ldots \subseteq F^\omega$ 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^0.

Note that ranks are often shifted by one, starting with $F = F^1$ or even by 2, starting with $F = F^2$.

Most examples in practice (and those we wrote) are in F^1, just above F.

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 : < get : 'a; .. >) : 'a = x#get
```

The dots ".." stands 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 alpha-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 become 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.

System F^ω For applicative functors

Generative ML modules (without parametric types) can be encoding in System F with existential types.

- A functor F has a type of the form: $\forall\alpha.\,\tau[\alpha] \to \exists\beta.\,.\sigma[\alpha,\beta]$
- If X, Y has type $\tau[\rho]$, then two successive applications F(X) and F(X) have types $\exists\beta.\,[\rho,\beta]$ with different abstract types β and cannot interoperate (on components involving β).

let Y = *unpack* $F\,X$ *in*
let Z = *unpack* $F\,X$ *in* is ill-typed
$Y = Z$

System F^ω For applicative functors

Generative ML modules (without parametric types) can be encoding in System F with existential types.

- A functor F has a type of the form: $\forall\alpha.\,\tau[\alpha] \to \exists\beta..\sigma[\alpha,\beta]$

However, *applicative* modules require the use of F^ω to keep track of type equalities! See [Rossberg et al., 2014] and [Rossberg, 2018].

- A functor F has a type of the form: $\exists\varphi.\,\forall\alpha.\,\tau[\alpha] \to \sigma[\alpha,\varphi\alpha]$ or when open $\forall\alpha.\,\tau[\alpha] \to \sigma[\alpha,\psi\rho]$ for some unknown ψ.
- Then if X has type $\tau[\rho]$, two successive applications F(X) and F(X) have the same type $\sigma[\rho,\varphi\rho]$ sharing the abstract type (application) $\psi\rho$.
- Hence, the two applications can interoperate,
- Key: ψ is abstract, which makes $\psi\rho$ abstract and incompatible with ρ, but all occurrences of $\psi\rho$ are compatible.

let ψ, f = *unpack* F *in*
let $Y = F\,X$ *in let* $Z = F\,X$ *in* $Y = Z$

is well-typed

System F^ω in OCamll

Second-order polymorphism in OCaml

- Via polymorphic methods

```
let id = object method f : 'a. 'a → 'a = fun x → x end
let y (x : <f : 'a. 'a → 'a>) = x#f x in y id
```

System F^ω in OCamll

Second-order polymorphism in OCaml

- Via polymorphic methods

```
let id = object method f : 'a. 'a → 'a = fun x → x end
let y (x : <f : 'a. 'a → 'a>) = x#f x in y id
```

- Via first-class modules

```
module type S = sig val f : 'a → 'a 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
```

System F^ω in OCamll

Second-order polymorphism in OCaml

- Via polymorphic methods
- Via first-class modules

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 now accessible 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]
}
```

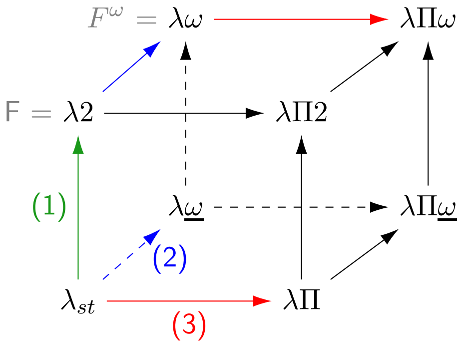
See https://www.baeldung.com/scala/dotty-scala-3

Still, this feature of Scala-3 is not emphrasized

- It was not directly accessible in previous version 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, *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
- They can be used to prove type safety and strong normalization

Binary relations

- Binary relations relates two 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 $\forall\alpha.\,\alpha \to \mathit{int}$?

?

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

so ?

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

for example ?

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.

What do they all have in common ?

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 $\forall\alpha.\,\alpha \rightarrow \mathit{int}$?

- 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 a term of the form $\lambda x.\,n$

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 $\lambda x. n$

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 $\forall\alpha.\,\alpha \to \alpha$?

?

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 $\lambda x.\, x$

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 $\forall\alpha\beta.\,\alpha \to \beta \to \alpha$?

?

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 $\forall\alpha.\,\alpha \to \alpha \to \alpha$?

?

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 $\forall \alpha. \alpha \to$ *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

$$\texttt{whoami} \;:\; \forall \alpha.\, \mathit{list}\,\alpha \to \mathit{list}\,\alpha$$

?

Pametricity

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

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

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

$$\texttt{whoami} \;:\; \forall \alpha.\, \textit{list}\,\alpha \to \textit{list}\,\alpha$$

- 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.

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

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?

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

?

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 $map\ f$

$$(\forall x, y,\ \ \mathit{cmp}_2\ (f\ x)\ (f\ y) = \mathit{cmp}_1\ \ x\ y) \implies \\ \forall \ell,\ \ \mathit{sort}\ \ \mathit{cmp}_2\ (\mathit{map}\ f\ \ell) = \mathit{map}\ f\ (\mathit{sort}\ \mathit{cmp}_1\ \ell)$$

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.

Can you give a few?

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.

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.

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.

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].

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:

$$
\begin{aligned}
\mathcal{V}[\![\mathsf{B}]\!] &\triangleq \{\mathsf{tt}, \mathsf{ff}\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!] &\triangleq \{\lambda x{:}\tau_1.\, M \mid \lambda x{:}\tau_1.\, M : \tau_1 \to \tau_2 \\
&\qquad\qquad \wedge\ \forall V \in \mathcal{V}[\![\tau_1]\!],\ (\lambda x{:}\tau_1.\, M)\, V \in \mathcal{E}[\![\tau_2]\!]\} \\
\mathcal{E}[\![\tau]\!] &\triangleq \{M \mid M : \tau \wedge \exists V \in \mathcal{V}[\![\tau]\!], M \Downarrow V\}
\end{aligned}
$$

We write $M \Downarrow V$ for $M \longrightarrow^* V$.
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.

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 $M \Downarrow V$ for $M \longrightarrow^* V$.
The goal is to show that any closed expression of type τ is in $\mathcal{E}[\![\tau]\!]$.

Remarks
$\mathcal{V}[\![\tau]\!] \subseteq \mathcal{E}[\![\tau]\!]$—by definition.
$\mathcal{E}[\![\tau]\!]$ is closed by inverse reduction—by definition, *i.e.*
If $M \longrightarrow N$ 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 $(\gamma M_1)\ (\gamma M_2) \rightsquigarrow (\gamma M_1)\ V \in \mathcal{E}[\![\tau]\!]$ 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

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:

$$\mathcal{E}[\![\tau]\!] = \left\{M : \tau \mid \forall N, M \Downarrow N \implies \left(N \in \mathcal{V}[\![\tau]\!] \ \vee\ \exists N', N \longrightarrow N'\right)\right\}$$

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

?

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

?

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. Assume $\Gamma \vdash M \equiv N : \tau$ holds and show that $\Gamma \vdash M \cong N : \tau$ holds (1).
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 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

$$
\begin{aligned}
\mathcal{V}[\![\mathsf{B}]\!] &\triangleq \{(\mathsf{tt},\mathsf{tt}),(\mathsf{ff},\mathsf{ff})\} \\
\mathcal{V}[\![\tau \to \sigma]\!] &\triangleq \{(V_1,V_2) \mid V_1,V_2 \vdash \tau \to \sigma \wedge \\
&\qquad \forall (W_1,W_2) \in \mathcal{V}[\![\tau]\!],\ (V_1\,W_1, V_2\,W_2) \in \mathcal{E}[\![\sigma]\!] \} \\
\mathcal{E}[\![\tau]\!] &\triangleq \{(M_1,M_2) \mid M_1,M_2 : \tau \wedge \\
&\qquad \exists (V_1,V_2) \in \mathcal{V}[\![\tau]\!],\ M_1 \Downarrow V_1 \wedge M_2 \Downarrow V_2\}
\end{aligned}
$$

In the following we will leave the typing constraint in gray implicit (as 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

We recursively define two relations $\mathcal{V}[\![\tau]\!]$ and $\mathcal{E}[\![\tau]\!]$ between values of type τ and expressions of type τ by

$$\begin{array}{rcl} \mathcal{V}[\![\mathsf{B}]\!] & \triangleq & \{(\mathsf{tt},\mathsf{tt}),(\mathsf{ff},\mathsf{ff})\} \\ \mathcal{V}[\![\tau \to \sigma]\!] & \triangleq & \{(V_1,V_2) \mid V_1,V_2 \vdash \tau \to \sigma \wedge \\ & & \quad \forall (W_1,W_2) \in \mathcal{V}[\![\tau]\!],\ (V_1\ W_1, V_2\ W_2) \in \mathcal{E}[\![\sigma]\!]\ \} \\ \mathcal{E}[\![\tau]\!] & \triangleq & \{(M_1,M_2) \mid M_1,M_2 : \tau \wedge \\ & & \quad \Downarrow (M_1,M_2) \in \mathcal{V}[\![\tau]\!]\ \} \end{array}$$

In the following we will leave the typing constraint in gray implicit (as 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

$$
\begin{aligned}
\mathcal{V}[\![\tau \to \sigma]\!] \;\triangleq\;& \{(V_1, V_2) \mid V_1, V_2 \vdash \tau \to \sigma \wedge \\
& \quad \forall (W_1, W_2) \in \mathcal{V}[\![\tau]\!],\ (V_1\ W_1, V_2\ W_2) \in \mathcal{E}[\![\sigma]\!]\} \\
=\;& \{((\lambda x{:}\tau.\, M_1), (\lambda x{:}\tau.\, M_2)) \mid (\lambda x{:}\tau.\, M_1), (\lambda x{:}\tau.\, M_2) \vdash \tau \to \sigma \wedge \\
& \quad \forall (W_1, W_2) \in \mathcal{V}[\![\tau]\!],\ ((\lambda x{:}\tau.\, M_1)\ W_1, (\lambda x{:}\tau.\, M_2)\ W_2) \in \mathcal{E}[\![\sigma]\!]\}
\end{aligned}
$$

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.

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 $V_2\ M_1 \sim_{\tau_2} V_1\ W_2$ (C). We have $W_2 \approx_{\tau_1} W_1$ by symmetry at type τ. By (H), we have $V_2\ W_2 \sim_{\tau_2} V_1\ W_1$ and (C) follows by symmetry of $\sim$ at type σ.

For transitivity, assume $V_1 \approx_\tau V_2$ (H1) and $V_2 \approx_\tau V_3$ (H2). To show $V_1 \approx_\tau V_3$, 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 $V_1\ W_1 \sim_{\tau_2} V_2\ W_3$ (C1).

By *symmetry and transitivity* of $\approx_\tau$, we get $W_3 \approx_\tau W_3$. (not reflexivity!)

By (H2), we have $V_2\ W_3 \sim_\sigma V_3\ W_3$ (C2).
(C) follows by transitivity of $\sim_\sigma$ (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 bisubstitutions γ mapping variable type assignments to pairs of related values.

$$\begin{array}{rcl} \mathcal{G}[\![\varnothing]\!] & \triangleq & \{\varnothing\} \\ \mathcal{G}[\![\Gamma, x : \tau]\!] & \triangleq & \{\gamma, x \mapsto (V_1, V_2) \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \land (V_1, V_2) \in \mathcal{V}[\![\tau]\!]\} \end{array}$$

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

$$\textsc{C-True}\quad \Gamma \vdash \mathsf{tt} : \mathit{bool} \qquad \textsc{C-False}\quad \Gamma \vdash \mathsf{ff} : \mathit{bool} \qquad \textsc{C-Var}\quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\textsc{C-Abs}\quad \frac{\Gamma, x : \tau \vdash M_1 : \sigma}{\Gamma \vdash \lambda x{:}\tau.\, M_1 : \tau \to \sigma} \qquad \textsc{C-App}\quad \frac{\Gamma \vdash M_1 : \tau \to \sigma \qquad \Gamma \vdash N_1 : \tau}{\Gamma \vdash M_1\ N_1 : \sigma}$$

$$\textsc{C-If}\quad \frac{\Gamma \vdash M_1 : \mathsf{B} \qquad \Gamma \vdash N_1 : \tau \qquad \Gamma \vdash N_1' : \tau}{\Gamma \vdash \mathsf{if}\ M_1\ \mathsf{then}\ N_1\ \mathsf{else}\ N_1' : \tau}$$

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
$$\Gamma \vdash \mathsf{tt} : \textit{bool}$$

C-False
$$\Gamma \vdash \mathsf{ff} : \textit{bool}$$

C-Var
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

C-Abs
$$\frac{\Gamma, x : \tau \vdash M_1 : \sigma}{\Gamma \vdash \lambda x{:}\tau.\, M_1 : \tau \to \sigma}$$

C-App
$$\frac{\Gamma \vdash M_1 : \tau \to \sigma \qquad \Gamma \vdash N_1 : \tau}{\Gamma \vdash M_1\ N_1 : \sigma}$$

C-If
$$\frac{\Gamma \vdash M_1 : \mathsf{B} \qquad \Gamma \vdash N_1 : \tau \qquad \Gamma \vdash N_1' : \tau}{\Gamma \vdash \mathsf{if}\ M_1\ \mathsf{then}\ N_1\ \mathsf{else}\ N_1' : \tau}$$

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
$$\Gamma \vdash \mathsf{tt} \quad : \textit{bool}$$

C-False
$$\Gamma \vdash \mathsf{ff} \quad : \textit{bool}$$

C-Var
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \quad : \tau}$$

C-Abs
$$\frac{\Gamma, x : \tau \vdash M_1 \quad : \sigma}{\Gamma \vdash \lambda x{:}\tau.\, M_1 \quad : \tau \to \sigma}$$

C-App
$$\frac{\Gamma \vdash M_1 \quad : \tau \to \sigma \qquad \Gamma \vdash N_1 \quad : \tau}{\Gamma \vdash M_1\ N_1 \quad : \sigma}$$

C-If
$$\frac{\Gamma \vdash M_1 \quad : \mathsf{B} \qquad \Gamma \vdash N_1 \quad : \tau \qquad \Gamma \vdash N_1' \quad : \tau}{\Gamma \vdash \mathsf{if}\ M_1\ \mathsf{then}\ N_1\ \mathsf{else}\ N_1' \quad : \tau}$$

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

$$\frac{}{\Gamma \vdash \mathsf{tt} \sim \mathsf{tt} : \mathit{bool}}\ \textsc{C-True} \qquad \frac{}{\Gamma \vdash \mathsf{ff} \sim \mathsf{ff} : \mathit{bool}}\ \textsc{C-False} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x \sim x : \tau}\ \textsc{C-Var}$$

$$\frac{\Gamma, x : \tau \vdash M_1 \sim M_2 : \sigma}{\Gamma \vdash \lambda x{:}\tau.\, M_1 \sim \lambda x{:}\tau.\, M_2 : \tau \to \sigma}\ \textsc{C-Abs} \qquad \frac{\Gamma \vdash M_1 \sim M_2 : \tau \to \sigma \qquad \Gamma \vdash N_1 \sim N_2 : \tau}{\Gamma \vdash M_1\, N_1 \sim M_2\, N_2 : \sigma}\ \textsc{C-App}$$

$$\frac{\Gamma \vdash M_1 \sim M_2 : \mathsf{B} \qquad \Gamma \vdash N_1 \sim N_2 : \tau \qquad \Gamma \vdash N_1' \sim N_2' : \tau}{\Gamma \vdash \mathsf{if}\ M_1\ \mathsf{then}\ N_1\ \mathsf{else}\ N_1' \sim \mathsf{if}\ M_2\ \mathsf{then}\ N_2\ \mathsf{else}\ N_2' : \tau}\ \textsc{C-If}$$

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
$$\Gamma \vdash \mathsf{tt} \sim \mathsf{tt} : \textit{bool}$$

C-False
$$\Gamma \vdash \mathsf{ff} \sim \mathsf{ff} : \textit{bool}$$

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
$$\frac{\Gamma \vdash M_1 \sim M_2 : \mathsf{B} \qquad \Gamma \vdash N_1 \sim N_2 : \tau \qquad \Gamma \vdash N_1' \sim N_2' : \tau}{\Gamma \vdash \mathsf{if}\ M_1\ \mathsf{then}\ N_1\ \mathsf{else}\ N_1' \sim \mathsf{if}\ M_2\ \mathsf{then}\ N_2\ \mathsf{else}\ N_2' : \tau}$$

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 $\mathcal{V}[\![\tau]\!]$. It suffices to show that $(\gamma_1(\lambda x{:}\tau.\, M_1)\ V_1, \gamma_2(\lambda x{:}\tau.\, M_2)\ V_2) \in \mathcal{E}[\![\sigma]\!]$ $(\mathbf{2})$.

Let γ' be $\gamma, x \mapsto (V_1, V_2)$. We have $\gamma' \in \mathcal{G}[\![\Gamma, x : \tau]\!]$. Thus, from (1), we have $(\gamma'_1 M_1, \gamma'_2 M_2) \in \mathcal{E}[\![\sigma]\!]$, which proves (2), since $\mathcal{E}[\![\sigma]\!]$ is closed by inverse reduction and $\gamma_1(\lambda x{:}\tau.\, M_1)\ V_1 \Downarrow \gamma'_i M_i$.

Rule C-APP *(and* C-IF*)*: By induction hypothesis and the fact that substitution distribute 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.* $\gamma_i(\text{if } M_i \text{ then } N_i \text{ else } N_i') \Downarrow \gamma_i N_i$. 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 proof 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 $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Delta \triangleright \sigma)$, then
$\Delta \vdash \mathcal{C}[M] \sim \mathcal{C}[M'] : \sigma$.

Proof By induction on the proof of $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Delta \triangleright \sigma)$.

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 it suffices to 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 $\mathbf{(4)}$. 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 $M \cong_{\mathsf{B}\to\tau\to\tau\to\tau} M'$.

?

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 $V_2 \approx_\tau V_2'$ $(\mathbf{3})$.

?

Logical equivalence: example of application

Fact: Assume $not \triangleq \lambda x{:}\mathsf{B}.\, \text{if } x \text{ then ff else tt}$
and $M \triangleq \lambda x{:}\mathsf{B}.\, \lambda y{:}\tau.\, \lambda z{:}\tau.\, \text{if } not\ x \text{ then } y \text{ else } z$
and $M' \triangleq \lambda x{:}\mathsf{B}.\, \lambda y{:}\tau.\, \lambda z{:}\tau.\, \text{if } x \text{ then } z \text{ else } y$.

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: $\text{if } not\ V_0 \text{ then } V_1 \text{ else } V_2 \ \sim_\tau\ \text{if } V_0' \text{ then } V_2' \text{ else } V_1'$ $(\mathbf{4})$.

?

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 $\mathcal{C} : (\Delta;\Gamma \triangleright \tau) \rightsquigarrow (\Delta';\Gamma' \triangleright \tau')$.

Observational equivalence

We (re)defined $\Delta;\Gamma \vdash M \cong M' : \tau$ as

$$\forall \mathcal{C} : (\Delta;\Gamma \triangleright \tau) \rightsquigarrow (\varnothing;\varnothing \triangleright \mathsf{B}),\ \ \mathcal{C}[M] \simeq \mathcal{C}[M']$$

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.* which are *closed by observational equivalence*:

$$R \in \mathcal{R}(\tau_1, \tau_2) \implies \forall (V_1, V_2) \in R,\ \forall W_1, W_2,\ W_1 \cong V_1 \wedge W_2 \cong V_2 \implies (W_1, W_2) \in R$$

The *restriction to admissible relations is required for completeness* of logical equivalence with respect to observational equivalence (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 $\mathcal{R}(\mathsf{B}, \mathit{int})$.

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.

Why?

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 $\mathcal{R}(\mathsf{B}, \mathit{int})$.

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.

Indeed, 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.

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 $\mathcal{R}(\mathsf{B}, \mathit{int})$.

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.

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 for mappings from type variables to such triples and 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

$$\mathcal{V}[\![\alpha]\!]_\eta \triangleq \eta_R(\alpha)$$

$$\mathcal{V}[\![\forall\alpha.\,\tau]\!]_\eta \triangleq \{(V_1, V_2) \mid V_1 : \eta_1(\forall\alpha.\,\tau) \wedge V_2 : \eta_2(\forall\alpha.\,\tau) \wedge \forall \rho_1, \rho_2, \forall R \in \mathcal{R}(\rho_1, \rho_2), (V_1\ \rho_1, V_2\ \rho_2) \in \mathcal{E}[\![\tau]\!]_{\eta, \alpha \mapsto (\rho_1, \rho_2, R)}\}$$

Logical equivalence for closed terms with open types

We redefine

$$
\begin{aligned}
\mathcal{V}[\![\mathsf{B}]\!]_{\eta} &\triangleq \{(\mathsf{tt},\mathsf{tt}),(\mathsf{ff},\mathsf{ff})\} \\
\mathcal{V}[\![\tau \to \sigma]\!]_{\eta} &\triangleq \{(V_1,V_2) \mid V_1 \vdash \eta_1(\tau \to \sigma) \wedge V_2 \vdash \eta_2(\tau \to \sigma) \wedge \\
&\qquad \forall (W_1,W_2) \in \mathcal{V}[\![\tau]\!]_{\eta},\ (V_1\ W_1, V_2\ W_2) \in \mathcal{E}[\![\sigma]\!]_{\eta}\} \\
\mathcal{E}[\![\tau]\!]_{\eta} &\triangleq \{(M_1,M_2) \mid M_1 : \eta_1\tau \wedge M_2 : \eta_2\tau \wedge \\
&\qquad \exists (V_1,V_2) \in \mathcal{V}[\![\tau]\!]_{\eta}, M_1 \Downarrow V_1 \wedge M_2 \Downarrow V_2\} \\
\mathcal{G}[\![\varnothing]\!]_{\eta} &\triangleq \{\varnothing\} \\
\mathcal{G}[\![\Gamma, x:\tau]\!]_{\eta} &\triangleq \{\gamma, x \mapsto (V_1,V_2) \mid \gamma \in \mathcal{G}[\![\Gamma]\!]_{\eta} \wedge (V_1,V_2) \in \mathcal{V}[\![\tau]\!]_{\eta}\}
\end{aligned}
$$

and define

$$
\begin{aligned}
\mathcal{D}[\![\varnothing]\!] &\triangleq \{\varnothing\} \\
\mathcal{D}[\![\Delta,\alpha]\!] &\triangleq \{\eta, \alpha \mapsto (\rho_1,\rho_2,\mathcal{R}) \mid \eta \in \mathcal{D}[\![\Delta]\!] \wedge R \in \mathcal{R}(\rho_1,\rho_2)\}
\end{aligned}
$$

Logical equivalence for open terms

Definition We define $\Delta; \Gamma \vdash M \sim M' : \tau$ as

$$\wedge \left\{ \begin{array}{l} \Delta; \Gamma \vdash M, M' : \tau \\ \forall \eta \in \mathcal{D}[\![\Delta]\!],\ \forall \gamma \in \mathcal{G}[\![\Gamma]\!]_\eta,\ (\eta_1(\gamma_1 M_1), \eta_2(\gamma_2 M_2)) \in \mathcal{E}[\![\tau]\!]_\eta \end{array} \right.$$

(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.

Other cases: the proof is similar to the case of the simply-typed λ-calculus.

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.

Corollary

The relation $\mathcal{V}[\![\tau]\!]^\sharp_\eta$ is an admissible relation in $\mathcal{R}^\sharp(\eta_1\tau, \eta_2\tau)$.

Application: we may take this relation when admissibility is required.

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 $\eta \in \mathcal{D}[\![\Delta]\!]$. Let R be $\mathcal{V}[\![\sigma]\!]_\eta$. Then,

$$\mathcal{V}[\![\tau[\alpha \mapsto \sigma]]\!]_\eta \quad = \quad \mathcal{V}[\![\tau]\!]_{\eta, \alpha \mapsto (\eta_1 \sigma, \eta_2 \sigma, R)}$$

Proof by structural 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 redefined the lemmas to work in a typing context of the form Δ, Γ instead of Γ and add two new lemmas:

$$\text{C-TABS}\quad \frac{\Delta, \alpha; \Gamma \vdash M_1 \sim M_2 : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.\, M_1 \sim \Lambda\alpha.\, M_2 : \forall\alpha.\,\tau}$$

$$\text{C-TAPP}\quad \frac{\Delta; \Gamma \vdash M_1 \sim M_2 : \forall\alpha.\,\tau \qquad \Delta \vdash \sigma}{\Delta; \Gamma \vdash M_1\ \sigma \sim M_2\ \sigma : \tau[\alpha \mapsto \sigma]}$$

Properties

Soundness of logical equivalence

Logical equivalence implies 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 $\Delta; \Gamma \vdash M_1 \sim^\sharp M_2 : \tau$.

Note: Admissibility is required for completeness, but not for soundness.

As a particular case, $M_1 \sim^\sharp_\tau M_2$ iff $M_1 \cong_\tau M_2$.

Properties

Extensionality *(Uses but 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:

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).

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 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.)

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 $[\alpha \mapsto (\rho,\rho,R)]$. 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).

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**).

?

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 $\alpha \mapsto (\mathsf{B}, \rho, R)$.

?

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 $M\ \rho\ V_1\ V_2 \cong_\sigma V_i$ $(\mathbf{1})$.

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.

?

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, $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\ V_1\ V_2)$ 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:

$$\bigvee \begin{cases} M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{tt} & \wedge \quad M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{ff} & \wedge \quad M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

Next ?

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, $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\ V_1\ V_2)$ 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:

$$\forall \rho, V_1, V_2, \quad \bigvee \begin{cases} M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_\mathsf{B} \mathsf{tt} & \wedge \quad M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_\mathsf{B} \mathsf{ff} & \wedge \quad M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

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, $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\ V_1\ V_2)$ 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:

$$\bigvee \begin{cases} \forall \rho, V_1, V_2,\ \ M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{tt} \ \wedge\ M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ \forall \rho, V_1, V_2,\ \ M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{ff} \ \wedge\ M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

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 $M\ \rho\ V_1\ V_2 \cong_\sigma V_i$ $(\mathbf{1})$.

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, $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\ V_1\ V_2)$ 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:

$$\bigvee \begin{cases} M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{tt} & \wedge \quad M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{ff} & \wedge \quad M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

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, $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\ V_1\ V_2)$ 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:

$$\bigvee \begin{cases} M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{tt} & \wedge \quad M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{ff} & \wedge \quad M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

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 $M\ \rho\ V_1\ V_2 \cong_\sigma V_i$ ($\mathbf{1}$).

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, $(M\ \mathbb{N}\ \mathbf{0}\ \mathbf{1}, M\ \rho\ V_1\ V_2)$ 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:

$$\bigvee \begin{cases} M\ \mathbb{N}\ \mathbf{0}\ \mathbf{1} \cong_{\mathbb{N}} \mathbf{0} & \wedge\quad M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ M\ \mathbb{N}\ \mathbf{0}\ \mathbf{1} \cong_{\mathbb{N}} \mathbf{1} & \wedge\quad M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

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, $(M\ \sigma\ \ W_1\ W_2, M\ \rho\ V_1\ V_2)$ 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:

$$\bigvee \begin{cases} M\ \sigma\ \ W_1\ W_2 \cong_\sigma\ \ W_1 & \wedge\ \ M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ M\ \sigma\ \ W_1\ W_2 \cong_\sigma\ \ W_2 & \wedge\ \ M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

Since, $M\ \sigma\ \ W_1\ W_2$ is independent of ρ, V_1, and V_2, this actually shows (1).

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, $(M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}, M\ \rho\ V_1\ V_2)$ 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:

$$\bigvee \begin{cases} M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{tt} & \wedge\quad M\ \rho\ V_1\ V_2 \cong_\rho V_1 \\ M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff} \cong_{\mathsf{B}} \mathsf{ff} & \wedge\quad M\ \rho\ V_1\ V_2 \cong_\rho V_2 \end{cases}$$

Since, $M\ \mathsf{B}\ \mathsf{tt}\ \mathsf{ff}$ is independent of ρ, V_1, and V_2, this actually shows (1).

Exercise

Inhabitants of $\forall\alpha.\,\alpha \to \alpha$

Redo the proof that all inhabitants of of $\forall\alpha.\,\alpha \to \alpha \to \alpha$ are observationally equivalent to the identity, following the schema that we used for booleans.

Applications

Inhabitants of $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

?

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 $M \cong_{\textit{nat}} N_n$ 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.

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

?

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 $M : nat$, then $M \cong_{nat} N_n$ 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 $V_1^n\;V_2$.

?

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 $\{(W_1, W_2) \mid \exists k \in \mathbb{N}, S^k\ Z \cong_{\mathit{nat}} W_1 \wedge V_1^k\ V_2 \cong_\rho W_2\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(Z, V_2)$ (reduce both sides for $k = 0$).
We also have $(S, V_1) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$.

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 $M : \textit{nat}$, then $M \cong_{\textit{nat}} N_n$ for some integer n, where $N_n \triangleq \Lambda\alpha.\, \lambda f{:}\alpha \to \alpha.\, \lambda x{:}\alpha.\, f^n\, x$.

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 $\mathcal{R}(\textit{nat}, \rho)$ be $\{(W_1, W_2) \mid \exists k \in \mathbb{N}, S^k\ Z \cong_{\textit{nat}} W_1 \wedge V_1^k\ V_2 \cong_\rho W_2\}$ and η be $\alpha \mapsto (\textit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(Z, V_2)$ (reduce both sides for $k = 0$).
We also have $(S, V_1) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$. (A key to the proof.)

Indeed,

?

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 $\{(W_1, W_2) \mid \exists k \in \mathbb{N}, S^k\ Z \cong_{\mathit{nat}} W_1 \wedge V_1^k\ V_2 \cong_\rho W_2\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(Z, V_2)$ (reduce both sides for $k = 0$).
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 $\mathcal{V}[\![\alpha]\!]_\eta$, *i.e.* R. There exists k such that $W_1 \cong_{\mathit{nat}} S^k\ Z$ and $W_2 \cong_\rho V_1^k\ V_2$. By congruence $S\,W_1 \cong_{\mathit{nat}} S^{k+1}\ Z$ and $V_1\ W_2 \cong_\rho V_1^{k+1}\ V_2$. Since $(S^{k+1}\ Z, V_1^{k+1}\ V_2)$ is in $\mathcal{E}[\![\alpha]\!]_\eta$, so is $(S\,W_1, V_1\ W_2)$ by closure by observational equivalence.

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 $\{(W_1, W_2) \mid \exists k \in \mathbb{N}, S^k\ Z \simeq_{\mathit{nat}} W_1 \wedge V_1^k\ V_2 \simeq_\rho W_2\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(Z, V_2)$ (reduce both sides for $k = 0$).
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 $\{(W_1, W_2) \mid \exists k \in \mathbb{N}, S^k\,Z \cong_{\mathit{nat}} W_1 \wedge V_1^k\,V_2 \cong_\rho W_2\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(Z, V_2)$ (reduce both sides for $k = 0$).
We also have $(S, V_1) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$. (A key to the proof.)

By parametricity, we have $M \sim_{\mathit{nat}} M$.

?

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 $\{(W_1, W_2) \mid \exists k \in \mathbb{N}, S^k\ Z \simeq_{\mathit{nat}} W_1 \wedge V_1^k\ V_2 \simeq_\rho W_2\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(Z, V_2)$ (reduce both sides for $k = 0$).
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 $M\ \mathit{nat}\ S\ Z \simeq_{\mathit{nat}} S^n\ Z$ and $M\ \rho\ V_1\ V_2 \simeq_\rho V_1^n\ V_2$.

?

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 $\{(W_1, W_2) \mid \exists k \in \mathbb{N}, S^k\ Z \simeq_{\mathit{nat}} W_1 \wedge V_1^k\ V_2 \simeq_\rho W_2\}$ and η be $\alpha \mapsto (\mathit{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\![\alpha]\!]_\eta$ since $R(Z, V_2)$ (reduce both sides for $k = 0$).
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 $M\ \mathit{nat}\ S\ Z \simeq_{\mathit{nat}} S^n\ Z$ and $M\ \rho\ V_1\ V_2 \simeq_\rho V_1^n\ V_2$.

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 $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

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 as $L_0 \triangleq \{N\}$ 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

?

Applications $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

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 as $L_0 \triangleq \{N\}$ and

$$\mathcal{L}_{k+1} \triangleq \{C\,W_k\;N_k \mid W_k \in \mathsf{Val}(\tau) \wedge N_k \in \mathcal{L}_k\}$$

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 $\mathsf{Val}(\tau)$.

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 $\bigcup \mathcal{R}_n$ where $\mathcal{R}_{k+1}$ is $\{\Downarrow(C\,G\;T, V_2\;H\;U) \mid (G,H) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T,U) \in \mathcal{R}_k\}$ and $\mathcal{R}_0$ is $\{(N, V_1)\}$.

We have $(N, V_2) \in \mathcal{R}_0 \subseteq \mathcal{V}[\![\alpha]\!]_\eta$.
We also have $(C,\;V_2) \in \mathcal{V}[\![\tau \to \alpha \to \alpha]\!]_\eta$.

Applications $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

Fact Let τ be closed and *list* be $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$. If $M : \textit{list}$, then $M \cong_{\textit{list}} N_n$ for some N_n in $\mathcal{L}_n$ where $\mathcal{L}_k$ is defined inductively as $L_0 \triangleq \{N\}$ and $\mathcal{L}_{k+1} \triangleq \{C\, W_k\; N_k \mid W_k \in \mathrm{Val}(\tau) \land N_k \in \mathcal{L}_k\}$.

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 $C\ W_n\ (\ldots (C\ W_1\ N) \ldots)$ $(\mathbf{1})$.

Let ρ and $V_1 : \alpha \to \rho \to \rho$ and $V_2 : \rho$ be fixed. Let R in $\mathcal{R}(\textit{list}, \rho)$ be defined inductively as $\bigcup \mathcal{R}_n$ where $\mathcal{R}_{k+1}$ is $\{\Downarrow (C\, G\; T, V_2\; H\; U) \mid (G, H) \in \mathcal{V}[\![\tau]\!]_\eta \land (T, U) \in \mathcal{R}_k\}$ and $\mathcal{R}_0$ is $\{(N, V_1)\}$.

We have $(N, V_2) \in \mathcal{R}_0 \subseteq \mathcal{V}[\![\alpha]\!]_\eta$.

We also have $(C,\ V_2) \in \mathcal{V}[\![\tau \to \alpha \to \alpha]\!]_\eta$. (A key to the proof)

Indeed,

?

Applications $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

Fact Let τ be closed and *list* be $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$. If $M : \textit{list}$, then $M \cong_{\textit{list}} N_n$ for some N_n in $\mathcal{L}_n$ where $\mathcal{L}_k$ is defined inductively as $L_0 \triangleq \{\mathsf{N}\}$ and $\mathcal{L}_{k+1} \triangleq \{\mathsf{C}\, W_k\, N_k \mid W_k \in \mathsf{Val}(\tau) \wedge N_k \in \mathcal{L}_k\}$.

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 $\mathsf{C}\ W_n\ (\ldots(\mathsf{C}\ W_1\ \mathsf{N})\ldots)\ (\mathbf{1})$.

Let ρ and $V_1 : \alpha \to \rho \to \rho$ and $V_2 : \rho$ be fixed. Let R in $\mathcal{R}(\textit{list}, \rho)$ be defined inductively as $\bigcup \mathcal{R}_n$ where $\mathcal{R}_{k+1}$ is $\{\Downarrow (\mathsf{C}\, G\, T, V_2\ H\ U) \mid (G, H) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T, U) \in \mathcal{R}_k\}$ and $\mathcal{R}_0$ is $\{(\mathsf{N}, V_1)\}$.

We have $(\mathsf{N}, V_2) \in \mathcal{R}_0 \subseteq \mathcal{V}[\![\alpha]\!]_\eta$.

We also have $(\mathsf{C},\ V_2) \in \mathcal{V}[\![\tau \to \alpha \to \alpha]\!]_\eta$. (A key to the proof)

Indeed, assume (G, H) in $\mathcal{V}[\![\tau]\!]_\eta$ and (T, U) in $\mathcal{V}[\![\alpha]\!]_\eta$, *i.e.* in $\mathcal{R}_k$ for some k. Then, $\Downarrow (\mathsf{C}\, G\, T, V_2\ H\ U)$ is in $\mathcal{R}^{k+1} \subseteq \mathcal{V}[\![\alpha]\!]_\eta$. Hence, $(\mathsf{C}\, G\, T, V_2\ H\ U) \in \mathcal{E}[\![\alpha]\!]_\eta$, as expected.

Applications $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

Fact Let τ be closed and *list* be $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$. If $M : \textit{list}$, then $M \cong_{\textit{list}} N_n$ for some N_n in $\mathcal{L}_n$ where $\mathcal{L}_k$ is defined inductively as $L_0 \triangleq \{\mathsf{N}\}$ and $\mathcal{L}_{k+1} \triangleq \{\mathsf{C}\, W_k\ N_k \mid W_k \in \mathsf{Val}(\tau) \wedge N_k \in \mathcal{L}_k\}$.

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 $\mathsf{C}\ W_n\ (\ldots(\mathsf{C}\ W_1\ \mathsf{N})\ldots)\ (\mathbf{1})$.

Let ρ and $V_1 : \alpha \to \rho \to \rho$ and $V_2 : \rho$ be fixed. Let R in $\mathcal{R}(\textit{list}, \rho)$ be defined inductively as $\bigcup \mathcal{R}_n$ where $\mathcal{R}_{k+1}$ is $\{\Downarrow (\mathsf{C}\, G\ T, V_2\ H\ U) \mid (G, H) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T, U) \in \mathcal{R}_k\}$ and $\mathcal{R}_0$ is $\{(\mathsf{N}, V_1)\}$.

We have $(\mathsf{N}, V_2) \in \mathcal{R}_0 \subseteq \mathcal{V}[\![\alpha]\!]_\eta$. We also have $(\mathsf{C},\ V_2) \in \mathcal{V}[\![\tau \to \alpha \to \alpha]\!]_\eta$.

?

Applications $\quad \forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

Fact Let τ be closed and *list* be $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$. 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 as $L_0 \triangleq \{N\}$ 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 $C\;W_n\;(\ldots(C\;W_1\;N)\ldots)\;(\mathbf{1})$.

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 $\bigcup \mathcal{R}_n$ where $\mathcal{R}_{k+1}$ is $\{\Downarrow(C\,G\;T, V_2\;H\;U) \mid (G, H) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T, U) \in \mathcal{R}_k\}$ and $\mathcal{R}_0$ is $\{(N, V_1)\}$.

We have $(N, V_2) \in \mathcal{R}_0 \subseteq \mathcal{V}[\![\alpha]\!]_\eta$. We also have $(C,\;V_2) \in \mathcal{V}[\![\tau \to \alpha \to \alpha]\!]_\eta$.

By parametricity, we have $M \sim_{\mathit{list}} M$.

?

Applications $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

Fact Let τ be closed and *list* be $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$. If $M : \textit{list}$, then $M \cong_{\textit{list}} N_n$ for some N_n in $\mathcal{L}_n$ where $\mathcal{L}_k$ is defined inductively as $L_0 \triangleq \{N\}$ 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 $C\ W_n\ (\ldots(C\ W_1\ N)\ldots)\ (\mathbf{1})$.

Let ρ and $V_1 : \alpha \to \rho \to \rho$ and $V_2 : \rho$ be fixed. Let R in $\mathcal{R}(\textit{list}, \rho)$ be defined inductively as $\bigcup \mathcal{R}_n$ where $\mathcal{R}_{k+1}$ is $\{\Downarrow (C\,G\,T, V_2\ H\ U) \mid (G, H) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T, U) \in \mathcal{R}_k\}$ and $\mathcal{R}_0$ is $\{(N, V_1)\}$.

We have $(N, V_2) \in \mathcal{R}_0 \subseteq \mathcal{V}[\![\alpha]\!]_\eta$. We also have $(C,\ V_2) \in \mathcal{V}[\![\tau \to \alpha \to \alpha]\!]_\eta$.

By parametricity, we have $M \sim_{\textit{list}} M$. Hence, $(M\ \textit{list}\ C\ N, M\ \rho\ V_1\ V_2) \in \mathcal{E}[\![\alpha]\!]_\eta$. Thus, there exists n such that $M\ \textit{list}\ C\ N \cong_{\textit{list}} C\ W_n\ (\ldots(C\ W_1\ N)\ldots)$ and $M\ \rho\ V_1\ V_2 \cong_\rho V_2\ W_n\ (\ldots(V_2\ W_1\ V_1)\ldots)$.

?

Applications $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$

Fact Let τ be closed and *list* be $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$. 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 as $L_0 \triangleq \{N\}$ 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 $C\;W_n\;(\ldots(C\;W_1\;N)\ldots)$ $(\mathbf{1})$.

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 $\bigcup \mathcal{R}_n$ where $\mathcal{R}_{k+1}$ is $\{\Downarrow(C\,G\;T, V_2\;H\;U) \mid (G,H) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T,U) \in \mathcal{R}_k\}$ and $\mathcal{R}_0$ is $\{(N, V_1)\}$.

We have $(N, V_2) \in \mathcal{R}_0 \subseteq \mathcal{V}[\![\alpha]\!]_\eta$. We also have $(C,\;V_2) \in \mathcal{V}[\![\tau \to \alpha \to \alpha]\!]_\eta$.

By parametricity, we have $M \sim_{\mathit{list}} M$. Hence, $(M\;\mathit{list}\;C\;N, M\;\rho\;V_1\;V_2) \in \mathcal{E}[\![\alpha]\!]_\eta$. Thus, there exists n such that $M\;\mathit{list}\;C\;N \cong_{\mathit{list}} C\;W_n\;(\ldots(C\;W_1\;N)\ldots)$ and $M\;\rho\;V_1\;V_2 \cong_\rho V_2\;W_n\;(\ldots(V_2\;W_1\;V_1)\ldots)$.

Since, $M\;\mathit{list}\;C\;N$ is independent of n and $(W_k)_{k\in 1..n}$, we may conclude (1). (This uses that $\mathcal{R}_k$ are all in different observational equivalence classes, which is easy to check, as a length function would return different integers.)

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: uad $\mathcal{V}[\![\mathit{nat}]\!] \triangleq \{(n, n) \mid n \in \mathbb{N}\}$

All properties are preserved.

Encodable features

Products

Given closed types τ_1 and τ_2, we defined

$$\begin{array}{rcl} \tau_1 \times \tau_2 & \triangleq & \forall \alpha. (\tau_1 \to \tau_2 \to \alpha) \to \alpha \\ (M_1, M_2) & \triangleq & \Lambda \alpha. \lambda x{:}\tau_1 \to \tau_2 \to \alpha. x\ M_1\ M_2 \\ M.i & \triangleq & M\ (\lambda x_1{:}\tau_1. \lambda x_2{:}\tau_2. x_i) \end{array}$$

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,

$$\mathcal{V}[\![\tau \times \sigma]\!]_\eta \triangleq \big\{\big((V_1, W_1), (V_2, W_2)\big) \;\big|\; (V_1, V_2) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (W_1, W_2) \in \mathcal{V}[\![\sigma]\!]_\eta\big\}$$

Sums

We define:

$$\begin{aligned}\mathcal{V}[\![\tau + \sigma]\!]_\eta &= \{(\mathit{inj}_1\ V_1, \mathit{inj}_1\ V_2) \mid (V_1, V_2) \in \mathcal{V}[\![\tau]\!]_\eta\} \ \cup \\ &\ \{(\mathit{inj}_2\ V_2, \mathit{inj}_2\ V_2) \mid (V_1, V_2) \in \mathcal{V}[\![\sigma]\!]_\eta\}\end{aligned}$$

Notice that sums, as all datatypes, can also be encoded in System F.

Primitive Lists

We *recursively*[1] define

$$\mathcal{V}[\![\mathit{list}\,\tau]\!]_\eta \triangleq \bigcup_k \mathcal{V}_k$$

$$\begin{array}{llcl} \text{where} & \mathcal{V}_0 & = & \{(\mathit{Nil}, \mathit{Nil})\} \\ & \mathcal{V}_{k+1} & = & \{(\mathit{Cons}\,H_1\,T_1, \mathit{Cons}\,H_2\,T_2) \\ & & & \quad \mid\ (H_1, H_2) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T_1, T_2) \in \mathcal{V}_k\} \end{array}$$

[1] This definition is well-founded. We may also use step-indexed relations.

Primitive Lists

We *recursively*[1] define

$$\begin{array}{rcl} \mathcal{V}[\![\mathit{list}\,\tau]\!]_\eta & \triangleq & \bigcup_k \mathcal{V}_k \\ \text{where} \quad \mathcal{V}_0 & = & \{(\mathit{Nil}, \mathit{Nil})\} \\ \mathcal{V}_{k+1} & = & \{(\mathit{Cons}\,H_1\,T_1, \mathit{Cons}\,H_2\,T_2) \\ & & \quad \mid \ (H_1, H_2) \in \mathcal{V}[\![\tau]\!]_\eta \ \wedge \ (T_1, T_2) \in \mathcal{V}_k\} \end{array}$$

Let R in $\mathcal{R}(\rho_1, \rho_2)$ be the graph $\langle g \rangle$ of a function g, *i.e.* equal to $\{(x, y) \mid g\,x = y\}$ and η be $\eta(\tau \mapsto \rho_1, \rho_2, R)$. Then, we have:

$$\mathcal{V}[\![\mathit{list}\,\tau]\!]_\eta(y_1, y_2)$$

$$\triangleq \ \bigvee \begin{cases} y_1 = \mathit{Nil} \wedge y_2 = \mathit{Nil} \\ y_1 = \mathit{Cons}\,H_1\,T_1 \wedge \\ \qquad y_2 = \mathit{Cons}\,H_2\,T_2 \wedge g\,H_1 = H_2 \wedge (T_1, T_2) \in \mathcal{V}_k \end{cases}$$

[1] This definition is well-founded. We may also use step-indexed relations.

Primitive Lists

We *recursively*[1] define

$$\mathcal{V}[\![\textit{list}\,\tau]\!]_\eta \triangleq \textstyle\bigcup_k \mathcal{V}_k$$

$$\begin{array}{llcl} \text{where} & \mathcal{V}_0 & = & \{(\textit{Nil}, \textit{Nil})\} \\ & \mathcal{V}_{k+1} & = & \{(\textit{Cons}\,H_1\,T_1, \textit{Cons}\,H_2\,T_2) \\ & & & \quad \mid\ (H_1, H_2) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T_1, T_2) \in \mathcal{V}_k\} \end{array}$$

Let R in $\mathcal{R}(\rho_1, \rho_2)$ be the graph $\langle g \rangle$ of a function g, *i.e.* equal to $\{(x, y) \mid g\,x = y\}$ and η be $\eta(\tau \mapsto \rho_1, \rho_2, R)$. Then, we have:

$$\mathcal{V}[\![\textit{list}\,\tau]\!]_\eta(y_1, y_2)$$

$$\triangleq \bigvee \left\{ \begin{array}{l} y_1 = \textit{Nil} \wedge y_2 = \textit{Nil} \\ y_1 = \textit{Cons}\,H_1\,T_1 \wedge \\ \qquad y_2 = \textit{Cons}\,(g\,H_1)\,T_2 \wedge (T_1, T_2) \in \mathcal{V}_k \end{array} \right.$$

[1] This definition is well-founded. We may also use step-indexed relations.

Primitive Lists

We *recursively*[1] define

$$\begin{array}{lll} \mathcal{V}[\![\textit{list}\,\tau]\!]_\eta & \triangleq & \bigcup_k \mathcal{V}_k \\ \quad \text{where} & \mathcal{V}_0 & = \{(\textit{Nil}, \textit{Nil})\} \\ & \mathcal{V}_{k+1} & = \{(\textit{Cons}\ H_1\ T_1, \textit{Cons}\ H_2\ T_2) \\ & & \qquad \mid\ (H_1, H_2) \in \mathcal{V}[\![\tau]\!]_\eta \ \wedge\ (T_1, T_2) \in \mathcal{V}_k\} \end{array}$$

Let R in $\mathcal{R}(\rho_1, \rho_2)$ be the graph $\langle g \rangle$ of a function g, *i.e.* equal to $\{(x, y) \mid g\ x = y\}$ and η be $\eta(\tau \mapsto \rho_1, \rho_2, R)$. Then, we have:

$$\mathcal{V}[\![\textit{list}\,\tau]\!]_\eta(y_1, y_2)$$

$$\triangleq\ \bigvee \begin{cases} y_1 = \textit{Nil} \wedge y_2 = \textit{Nil} \\ y_1 = \textit{Cons}\ H_1\ T_1 \wedge \\ \qquad y_2 = \textit{Cons}\ (g\ H_1)\ T_2 \wedge (T_1, T_2) \in \mathcal{V}_k \end{cases}$$

$$\triangleq\ \textit{map}\ \rho_1\ \rho_2\ g\ \ y_1 \Downarrow y_2$$

[1] This definition is well-founded. We may also use step-indexed relations.

Primitive Lists

We *recursively*[1] define

$$
\begin{array}{lll}
\mathcal{V}[\![\mathit{list}\,\tau]\!]_\eta & \triangleq & \bigcup_k \mathcal{V}_k \\
\text{where} & \mathcal{V}_0 & = \{(\mathit{Nil}, \mathit{Nil})\} \\
& \mathcal{V}_{k+1} & = \{(\mathit{Cons}\, H_1\, T_1, \mathit{Cons}\, H_2\, T_2) \\
& & \quad \mid\ (H_1, H_2) \in \mathcal{V}[\![\tau]\!]_\eta \wedge (T_1, T_2) \in \mathcal{V}_k\}
\end{array}
$$

Let R in $\mathcal{R}(\rho_1, \rho_2)$ be the graph $\langle g\rangle$ of a function g, *i.e.* equal to $\{(x, y) \mid g\, x = y\}$ and η be $\eta(\tau \mapsto \rho_1, \rho_2, R)$. Then, we have:

$$\mathcal{V}[\![\mathit{list}\,\tau]\!]_\eta = \Downarrow \langle \mathit{map}\ \rho_1\ \rho_2\ g\ \rangle$$

[1] This definition is well-founded. We may also use step-indexed relations.

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

Fact: Assume *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 $\mathit{sort} : \forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha$

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

Fact: Assume *sort* $: \ \forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha \to \mathit{list}\,\alpha$ $(\mathbf{1})$. Then

$$(\forall x, y,\ \ \mathit{cmp}_2\ (f\ x)\ (f\ y) \cong \mathit{cmp}_1\ \ x\ y) \implies \forall \ell,\ \ \mathit{sort}\ \ \mathit{cmp}_2\ (\mathit{map}\ f\ \ell) \cong \mathit{map}\ f\ (\mathit{sort}\ \mathit{cmp}_1\ \ell)$$

Applications $\quad$ *sort* $: \forall\alpha.\,(\alpha \to \alpha \to \textit{bool}) \to \textit{list}\,\alpha$

Proof: We have $\textit{sort} \sim_\sigma \textit{sort}$ where σ is $\forall\alpha.\,(\alpha \to \alpha \to \textit{bool}) \to \textit{list}\,\alpha \to \textit{list}\,\alpha$.
Thus, for all ρ_1, ρ_2, and admissible relations R in $\mathcal{R}(\rho_1, \rho_2)$,

$$\forall (\textit{cp}_1, \textit{cp}_2) \in \mathcal{V}[\![\alpha \to \alpha \to \mathsf{B}]\!]_\eta, \tag{1}$$

$$\forall (V_1, V_2) \in \mathcal{V}[\![\textit{list}\,\alpha]\!]_\eta,\ \ (\textit{sort}\,\rho_1\ \textit{cp}_1\ \ V_1, \textit{sort}\,\rho_2\ \textit{cp}_2\ \ V_2) \in \mathcal{E}[\![\textit{list}\,\alpha]\!]_\eta) \tag{2}$$

where η is $\alpha \mapsto (\rho_1, \rho_2, R)$.

Applications $\quad$ *sort* $: \forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha$

Proof: We have *sort* $\sim_\sigma$ *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 admissible relations R in $\mathcal{R}(\rho_1, \rho_2)$,

$$\forall (\mathit{cp}_1, \mathit{cp}_2) \in \mathcal{V}[\![\alpha \to \alpha \to \mathsf{B}]\!]_\eta, \qquad \textbf{(1)}$$
$$\forall (V_1, V_2) \in \mathcal{V}[\![\mathit{list}\,\alpha]\!]_\eta,\ \ (\mathit{sort}\ \rho_1\ \mathit{cp}_1\ \ V_1, \mathit{sort}\ \rho_2\ \mathit{cp}_2\ \ V_2) \in \mathcal{E}[\![\mathit{list}\,\alpha]\!]_\eta) \qquad \textbf{(2)}$$

where η is $\alpha \mapsto (\rho_1, \rho_2, R)$.
We may choose R to be $\langle f \rangle$ for some f.
Then (1), which means

$$\forall (V, V') \in \langle f \rangle,\ \ \forall (W, W') \in \langle f \rangle,\ \ (\mathit{cp}_1\ V\ W, \mathit{cp}_2\ V'\ W') \in \mathcal{V}[\![\mathsf{B}]\!]$$

becomes

$$\forall V, W : \rho_1,\ \ \mathit{cp}_1\ V\ W \ \ \cong_{\mathsf{B}}\ \ \mathit{cp}_2\ (f\ V)\ (f\ W)$$

and

$$\mathcal{V}[\![\mathit{list}\,\alpha]\!]_\eta \ \ \triangleq\ \Downarrow \langle \mathit{map}\ \rho_1\ \rho_2\ f \rangle \ \ \subseteq\ \ \mathcal{V}[\![\rho_1]\!] \times \mathcal{V}[\![\rho_2]\!]$$

Applications $\quad$ *sort* $: \forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha$

Proof: 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 admissible relations R in $\mathcal{R}(\rho_1, \rho_2)$,

$$\forall (\mathit{cp}_1, \mathit{cp}_2) \in \mathcal{V}[\![\alpha \to \alpha \to \mathsf{B}]\!]_\eta, \tag{1}$$
$$\forall (V_1, V_2) \in \mathcal{V}[\![\mathit{list}\,\alpha]\!]_\eta,\ \ (\mathit{sort}\,\rho_1\,\mathit{cp}_1\ \ V_1, \mathit{sort}\,\rho_2\,\mathit{cp}_2\ \ V_2) \in \mathcal{E}[\![\mathit{list}\,\alpha]\!]_\eta) \tag{2}$$

where η is $\alpha \mapsto (\rho_1, \rho_2, R)$.
We may choose R to be $\langle f \rangle$ for some f.
Then (1), which means

$$\forall (V, V') \in \langle f \rangle,\ \ \forall (W, W') \in \langle f \rangle,\ \ (\mathit{cp}_1\ V\ W, \mathit{cp}_2\ V'\ W') \in \mathcal{V}[\![\mathsf{B}]\!]$$

becomes

$$\forall V, W : \rho_1,\ \ \mathit{cp}_1\ V\ W\ \ \cong_{\mathsf{B}}\ \ \mathit{cp}_2\ (f\ V)\ (f\ W)$$

and

$$\mathcal{V}[\![\mathit{list}\,\alpha]\!]_\eta\ \ \triangleq\ \Downarrow \langle \mathit{map}\,\rho_1\,\rho_2\,f \rangle\ \ \subseteq\ \ \mathcal{V}[\![\rho_1]\!] \times \mathcal{V}[\![\rho_2]\!]$$

Thus, (3) reads

$$\forall V : \mathit{list}\,\rho_1, V' : \mathit{list}\,\rho_2,$$
$$\mathit{map}\,\rho_1\,\rho_2\,f\ V \Downarrow\ \ V' \implies \mathit{sort}\,\rho_2\,\mathit{cp}_2\,V'\ \sim_{\mathit{list}\,\rho_2}\ \mathit{map}\,\rho_1\,\rho_2\,f\ (\mathit{sort}\,\rho_1\,\mathit{cp}_1\ \ V)$$

Applications $\quad$ *sort* : $\forall\alpha.\,(\alpha \to \alpha \to \mathit{bool}) \to \mathit{list}\,\alpha$

Proof: We have *sort* $\sim_\sigma$ *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 admissible relations R in $\mathcal{R}(\rho_1,\rho_2)$,

$$\forall (\mathit{cp}_1, \mathit{cp}_2) \in \mathcal{V}[\![\alpha \to \alpha \to \mathsf{B}]\!]_\eta, \qquad \textbf{(1)}$$
$$\forall (V_1, V_2) \in \mathcal{V}[\![\mathit{list}\,\alpha]\!]_\eta,\ \ (\mathit{sort}\,\rho_1\,\mathit{cp}_1\ \ V_1, \mathit{sort}\,\rho_2\,\mathit{cp}_2\ \ V_2) \in \mathcal{E}[\![\mathit{list}\,\alpha]\!]_\eta) \qquad \textbf{(2)}$$

where η is $\alpha \mapsto (\rho_1, \rho_2, R)$.
We may choose R to be $\langle f \rangle$ for some f.
Then (1), which means

$$\forall (V, V') \in \langle f \rangle,\ \ \forall (W, W') \in \langle f \rangle,\ \ (\mathit{cp}_1\ V\ W, \mathit{cp}_2\ V'\ W') \in \mathcal{V}[\![\mathsf{B}]\!]$$

becomes

$$\forall V, W : \rho_1,\ \ \mathit{cp}_1\ V\ W \ \ \cong_{\mathsf{B}}\ \ \mathit{cp}_2\ (f\ V)\ (f\ W)$$

and

$$\mathcal{V}[\![\mathit{list}\,\alpha]\!]_\eta \ \ \triangleq\ \Downarrow \langle \mathit{map}\ \rho_1\ \rho_2\ f \rangle \ \ \subseteq\ \ \mathcal{V}[\![\rho_1]\!] \times \mathcal{V}[\![\rho_2]\!]$$

Thus, (3) implies

$$\forall V : \mathit{list}\ \rho_1, V' : \mathit{list}\ \rho_2,$$
$$\mathit{map}\ \rho_1\ \rho_2\ f\ V \sim_{\rho_2} V' \implies \mathit{sort}\ \rho_2\ \mathit{cp}_2\ V' \ \sim_{\mathit{list}\,\rho_2}\ \mathit{map}\ \rho_1\ \rho_2\ f\ (\mathit{sort}\ \rho_1\ \mathit{cp}_1\ \ V)$$

Applications $\quad$ $sort : \forall\alpha.\,(\alpha \to \alpha \to bool) \to list\,\alpha$

Proof: We have $sort \sim_\sigma sort$ where σ is $\forall\alpha.\,(\alpha \to \alpha \to bool) \to list\,\alpha \to list\,\alpha$.
Thus, for all ρ_1, ρ_2, and admissible relations R in $\mathcal{R}(\rho_1, \rho_2)$,

$$\forall (cp_1, cp_2) \in \mathcal{V}[\![\alpha \to \alpha \to \mathsf{B}]\!]_\eta, \qquad \textbf{(1)}$$
$$\forall (V_1, V_2) \in \mathcal{V}[\![list\,\alpha]\!]_\eta,\ \ (sort\ \rho_1\ cp_1\ \ V_1, sort\ \rho_2\ cp_2\ \ V_2) \in \mathcal{E}[\![list\,\alpha]\!]_\eta) \qquad \textbf{(2)}$$

where η is $\alpha \mapsto (\rho_1, \rho_2, R)$.
We may choose R to be $\langle f \rangle$ for some f.
Then (1), which means

$$\forall (V, V') \in \langle f \rangle,\ \ \forall (W, W') \in \langle f \rangle,\ \ (cp_1\ V\ W, cp_2\ V'\ W') \in \mathcal{V}[\![\mathsf{B}]\!]$$

becomes

$$\forall V, W : \rho_1,\ \ cp_1\ V\ W \ \ \cong_{\mathsf{B}}\ \ cp_2\ (f\ V)\ (f\ W)$$

and

$$\mathcal{V}[\![list\,\alpha]\!]_\eta \ \ \triangleq\ \Downarrow \langle map\ \rho_1\ \rho_2\ f \rangle \ \ \subseteq\ \ \mathcal{V}[\![\rho_1]\!] \times \mathcal{V}[\![\rho_2]\!]$$

Thus, (3) implies

$$\forall V : list\ \rho_1,$$
$$sort\ \rho_2\ cp_2\ (map\ \rho_1\ \rho_2\ f\ V) \sim_{list\ \rho_2} map\ \rho_1\ \rho_2\ f\ (sort\ \rho_1\ cp_1\ \ V)$$

Applications $sort : \forall\alpha.\,(\alpha \to \alpha \to bool) \to list\ \alpha$

Proof: We have $sort \sim_\sigma sort$ where σ is $\forall\alpha.\,(\alpha \to \alpha \to bool) \to list\ \alpha \to list\ \alpha$.
Thus, for all ρ_1, ρ_2, and admissible relations R in $\mathcal{R}(\rho_1, \rho_2)$,

$$\forall (cp_1, cp_2) \in \mathcal{V}[\![\alpha \to \alpha \to \mathsf{B}]\!]_\eta, \qquad \textbf{(1)}$$
$$\forall (V_1, V_2) \in \mathcal{V}[\![list\ \alpha]\!]_\eta,\ \ (sort\ \rho_1\ cp_1\ \ V_1, sort\ \rho_2\ cp_2\ \ V_2) \in \mathcal{E}[\![list\ \alpha]\!]_\eta) \qquad \textbf{(2)}$$

where η is $\alpha \mapsto (\rho_1, \rho_2, R)$.
We may choose R to be $\langle f \rangle$ for some f.
Then (1), which means

$$\forall (V, V') \in \langle f \rangle,\ \ \forall (W, W') \in \langle f \rangle,\ \ (cp_1\ V\ W, cp_2\ V'\ W') \in \mathcal{V}[\![\mathsf{B}]\!]$$

becomes

$$\forall V, W : \rho_1,\ \ cp_1\ V\ W\ \ \cong_{\mathsf{B}}\ \ cp_2\ (f\ V)\ (f\ W)$$

and

$$\mathcal{V}[\![list\ \alpha]\!]_\eta\ \ \triangleq\ \Downarrow \langle map\ \rho_1\ \rho_2\ f \rangle\ \ \subseteq\ \ \mathcal{V}[\![\rho_1]\!] \times \mathcal{V}[\![\rho_2]\!]$$

Thus, (3) implies

$$\forall V : list\ \rho_1,$$
$$sort\ \rho_2\ cp_2\ (map\ \rho_1\ \rho_2\ f\ V)\ \cong_{list\ \rho_2}\ map\ \rho_1\ \rho_2\ f\ (sort\ \rho_1\ cp_1\ \ V)$$

Applications $\quad$ *whoami* : $\forall\alpha.\,$*list* $\alpha \rightarrow$ *list* α

Left as an exercise...

Existential types

We define:

$$\begin{aligned}\mathcal{V}[\![\exists\alpha.\,\tau]\!]_\eta \;\triangleq\; \big\{&(\textit{pack}\; V_1, \rho_1 \;\textit{as}\; \exists\alpha.\,\tau, \textit{pack}\; V_2, \rho_2 \;\textit{as}\; \exists\alpha.\,\tau) \mid \\ &\exists \rho_1, \rho_2, R \in \mathcal{R}(\rho_1, \rho_2),\;\; (V_1, V_2) \in \mathcal{E}[\![\tau]\!]_{\eta, \alpha \mapsto (\rho_1, \rho_2, R)}\big\}\end{aligned}$$

Compare with

$$\begin{aligned}\mathcal{V}[\![\forall\alpha.\,\tau]\!]_\eta \;=\; \big\{&(\Lambda\alpha.\,M_1, \Lambda\alpha.\,M_2) \mid \\ &\forall \rho_1, \rho_2, R \in \mathcal{R}(\rho_1, \rho_2), \\ &\quad ((\Lambda\alpha.\,M_1)\;\rho_1, (\Lambda\alpha.\,M_2)\;\rho_2) \in \mathcal{E}[\![\tau]\!]_{\eta, \alpha \mapsto (\rho_1, \rho_2, R)}\big\}\end{aligned}$$

Existential types Example

Consider $V_1 \triangleq (\mathit{not}, \mathsf{tt})$, and $V_2 \triangleq (\mathit{succ}, 0)$ and $\sigma \triangleq (\alpha \to \alpha) \times \alpha$.
Let $R \in \mathcal{R}(\mathit{bool}, \mathit{nat})$ be $\{(\mathsf{tt}, 2n), (\mathsf{ff}, 2n+1) \mid n \in \mathbb{N}\}$ and η be $\alpha \mapsto (\mathit{bool}, \mathit{nat}, R)$.

We have $(V_1, V_2) \in \mathcal{V}[\![\sigma]\!]_\eta$.

Hence, $(\mathit{pack}\, V_1, \mathit{bool}\ \mathit{as}\ \exists\alpha.\,\sigma,\ \mathit{pack}\, V_2, \mathit{nat}\ \mathit{as}\ \exists\alpha.\,\sigma) \in \mathcal{V}[\![\exists\alpha.\,\sigma]\!]$.

Existential types — Example

Consider $V_1 \triangleq (\mathit{not}, \mathsf{tt})$, and $V_2 \triangleq (\mathit{succ}, 0)$ and $\sigma \triangleq (\alpha \to \alpha) \times \alpha$.
Let $R \in \mathcal{R}(\mathit{bool}, \mathit{nat})$ be $\{(\mathsf{tt}, 2n), (\mathsf{ff}, 2n+1) \mid n \in \mathbb{N}\}$ and η be $\alpha \mapsto (\mathit{bool}, \mathit{nat}, R)$.

We have $(V_1, V_2) \in \mathcal{V}[\![\sigma]\!]_\eta$.

Hence, $(\mathit{pack}\ V_1, \mathit{bool}\ \mathit{as}\ \exists\alpha.\,\sigma,\ \mathit{pack}\ V_2, \mathit{nat}\ \mathit{as}\ \exists\alpha.\,\sigma) \in \mathcal{V}[\![\exists\alpha.\,\sigma]\!]$.

Proof of $\big((\mathit{not}, \mathsf{tt}), (\mathit{succ}, 0)\big) \in \mathcal{V}[\![(\alpha \to \alpha) \times \alpha]\!]_\eta$ **(1)**

?

Existential types Example

Consider $V_1 \triangleq (\mathit{not}, \mathrm{tt})$, and $V_2 \triangleq (\mathit{succ}, 0)$ and $\sigma \triangleq (\alpha \to \alpha) \times \alpha$.
Let $R \in \mathcal{R}(\mathit{bool}, \mathit{nat})$ be $\{(\mathrm{tt}, 2n), (\mathrm{ff}, 2n+1) \mid n \in \mathbb{N}\}$ and η be $\alpha \mapsto (\mathit{bool}, \mathit{nat}, R)$.

We have $(V_1, V_2) \in \mathcal{V}[\![\sigma]\!]_\eta$.

Hence, $(\mathit{pack}\; V_1, \mathit{bool}\; \mathit{as}\; \exists\alpha.\,\sigma,\;\; \mathit{pack}\; V_2, \mathit{nat}\; \mathit{as}\; \exists\alpha.\,\sigma) \in \mathcal{V}[\![\exists\alpha.\,\sigma]\!]$.

Proof of $\big((\mathit{not}, \mathrm{tt}), (\mathit{succ}, 0)\big) \in \mathcal{V}[\![(\alpha \to \alpha) \times \alpha]\!]_\eta$ $\mathbf{(1)}$

We have $(\mathrm{tt}, 0) \in \mathcal{V}[\![\alpha]\!]_\eta$, since $(\mathrm{tt}, 0) \in R$.
We also have $(\mathit{not}, \mathit{succ}) \in \mathcal{V}[\![\alpha \to \alpha]\!]_\eta$, which proves (1).

?

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 $\forall\alpha.\,\tau \rightarrow \sigma$ 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 $(M, M) \in \mathcal{E}[\![\forall\alpha.\,\tau \rightarrow \sigma]\!]_\eta$. 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 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:

$$
\begin{aligned}
\mathcal{V}_k[\![\mathsf{B}]\!]_\eta &= \{(\mathsf{tt},\mathsf{tt}),(\mathsf{ff},\mathsf{ff})\}\\
\mathcal{V}_k[\![\tau \to \sigma]\!]_\eta &= \{(V_1,V_2) \mid \forall j < k, \forall (W_1,W_2) \in \mathcal{V}_j[\![\tau]\!]_\eta,\\
&\qquad (V_1\ W_1, V_2\ W_2) \in \mathcal{E}_j[\![\sigma]\!]_\eta\}\\
\mathcal{V}_k[\![\alpha]\!]_\eta &= \eta_R(\alpha).k\\
\mathcal{V}_k[\![\forall\alpha.\,\tau]\!]_\eta &= \big\{(V_1,V_2) \mid \forall \rho_1,\rho_2, R \in \mathcal{R}^k(\rho_1,\rho_2), \forall j < k,\\
&\qquad (V_1\ \rho_1, V_2\ \rho_2) \in \mathcal{V}_j[\![\tau]\!]_{\eta,\alpha\mapsto(\rho_1,\rho_2,R)}\big\}\\
\mathcal{V}_k[\![\mu\alpha.\tau]\!]_\eta &= \mathcal{V}_{k-1}[\![[\alpha \mapsto \mu\alpha.\tau]\tau]\!]_\eta\\
\mathcal{E}_k[\![\tau]\!]_\eta &= \{(M_1,M_2) \mid \forall j < k, M_1 \Downarrow_j V_1\\
&\qquad \Longrightarrow \exists V_2, M_2 \Downarrow V_2 \wedge (V_1,V_2) \in \mathcal{V}_{k-j}[\![\tau]\!]_\eta\}
\end{aligned}
$$

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 of the scope of this course, but one may, for instance, read [Atkey, 2012].

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

▷ 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: $10.1007/978\text{-}\sigma_2 3\text{-}\sigma_2 642\text{-}\sigma_2 11957\text{-}\sigma_2 6\_8$.

▷ 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.

Bibliography III

- 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.

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.

Bibliography IV

▷ 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.

▷ 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: $10.1007/3\text{-}\sigma_2 540\text{-}\sigma_2 45927\text{-}\sigma_2 8\backslash\_11$.

Bibliography V

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.

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.

▷ 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 VI

▷ 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.

▷ 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.

▷ Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–283, January 1996.

Bibliography VII

- 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.
- 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.

Bibliography VIII

- 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, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 41–53, 2001.
- Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

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.

Bibliography IX

▷ Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

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.

▷ François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.

▷ 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.

Bibliography X

- 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.
- 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.
- 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 XI

- 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.
- 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: $10.1007/978\text{-}\sigma_2 3\text{-}\sigma_2 662\text{-}\sigma_2 46669\text{-}\sigma_2 8\_28$.

Bibliography XII

▷ 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.

▷ 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.

▷ Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179(1):1–18, 2002.

Bibliography XIII

- 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.
- 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.
- 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.

Bibliography XIV

- 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.
- 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.