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

Didier Rémy
Plan of the course

Metatheory of System F

ADTs, Recursive types, Existential types, GATDs

Going higher order with $F^\omega$!

Logical relations

Side effects, References, Value restriction
Metatheory of System F
Proofs

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

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

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

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

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

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

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

- Hence, types must be *checked* and ill-typed programs must be rejected.
What are they useful for?

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

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

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

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

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

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

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

The issues are safety, expressiveness, and type inference.
Typed, Sir! with better types.

In fact, Reynolds settles the debate:

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

Today, the question is more whether

- to stay with rather simple polymorphic types (ML, System F, or $F^{\omega}$).
- use more sophisticated types (dependent types, afine types, capabilities and ownership, effects, logical assertions, etc.), or
- even towards full program proofs!

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

- Simply-typed $\lambda$-calculus
- Type soundness for simply-typed $\lambda$-calculus
- Simple extensions: Pairs, sums, recursive functions
- Polymorphism
- Polymorphic $\lambda$-calculus
- Type soundness
- Type erasing semantics
Why $\lambda$-calculus?

In this course, the underlying programming language is the $\lambda$-calculus.

The $\lambda$-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 $\lambda$-calculus, most of our results can also be applied to other languages (Java, assembly language, etc.).
Simply typed $\lambda$-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* $F^\omega$

**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, $\alpha$-conversion, and substitutions

$\lambda x:\tau. M$ binds variable $x$ in $M$.

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

$$\text{fv}(x) \triangleq \{x\}$$

$$\text{fv}(\lambda x:\tau. M) \triangleq \text{fv}(M) \setminus \{x\}$$

$$\text{fv}(M_1 M_2) \triangleq \text{fv}(M_1) \cup \text{fv}(M_2)$$

We write $x \not\in M$ for $x \notin \text{fv}(M)$.

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 $y \not\in \text{fv}(M)$.

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:

\[ V ::= \lambda x: \tau. \, M | \ldots \]

The reduction relation \( M_1 \rightarrow M_2 \) is inductively defined:

\[ (\lambda x: \tau. \, M) \, V \rightarrow [x \mapsto V] \, M \]

Evaluation contexts are defined as follows:

\[ E ::= [] \, M | V \, [] | \ldots \]

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

An evaluation context of arbitrary depth can be defined as:

\[ \overline{E} ::= [] | E[\overline{E}] \]
Static semantics

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

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

where $\Gamma$ is a typing context.
Typing context, notations

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

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

To avoid confusion, we require $x \notin \text{dom}(\Gamma)$ 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 $\text{dom}(\Gamma)$ for the set of variables bound by $\Gamma$ and $x : \tau \in \Gamma$ to mean $x \in \text{dom}(\Gamma)$ and $\Gamma(x) = \tau$. 
Static semantics

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

\[ \text{VAR} \]
\[ \Gamma \vdash x : \Gamma(x) \]

\[ \text{ABS} \]
\[ \Gamma, x : \tau_1 \vdash M : \tau_2 \quad \frac{}{\Gamma \vdash \lambda x : \tau_1. M : \tau_1 \rightarrow \tau_2} \]

\[ \text{APP} \]
\[ \Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash M_2 : \tau_1 \quad \frac{}{\Gamma \vdash M_1 \ M_2 : \tau_2} \]

Notice that the specification is extremely simple.

In the simply-typed \( \lambda \)-calculus, the definition is \textit{syntax-directed}. This is not true of all type systems.
Example

The following is a valid *typing derivation*:

\[
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash f : \tau \to \tau' \\
\text{APP} \quad \Gamma \vdash f \, x_1 : \tau' \\
\text{VAR} \quad \Gamma \vdash x_1 : \tau \\
\hline
\Gamma \vdash f \, x_1 : \tau' \\
\end{array}
\quad
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash x_2 : \tau \\
\text{APP} \quad \Gamma \vdash f \, x_2 : \tau' \\
\hline
\Gamma \vdash f \, x_2 : \tau' \\
\end{array}
\quad
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash x_2 : \tau \\
\text{APP} \quad \Gamma \vdash f \, x_2 : \tau' \\
\hline
\Gamma \vdash f \, x_2 : \tau' \\
\end{array}
\quad
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash x_1 : \tau \\
\hline
\Gamma \vdash f \, x_1 : \tau' \\
\end{array}
\quad
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash x_2 : \tau \\
\hline
\Gamma \vdash f \, x_2 : \tau' \\
\end{array}
\quad
\begin{array}{c}
\text{VAR} \quad \Gamma \vdash x_2 : \tau \\
\hline
\Gamma \vdash f \, x_2 : \tau' \\
\end{array}
\quad
\begin{array}{c}
\text{PAIR} \quad \Gamma \vdash (f \, x_1, f \, x_2) : \tau' \times \tau' \\
\hline
\end{array}
\quad
\begin{array}{c}
\text{ABS} \quad \emptyset \vdash \lambda f : \tau \to \tau'. \lambda x_1 : \tau. \lambda x_2 : \tau. (f \, x_1, f \, x_2) : (\tau \to \tau') \to \tau \to \tau \to (\tau' \times \tau') \\
\hline
\end{array}
\]

\(\Gamma\) 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 \(\tau\) and \(\tau'\)

(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 $x \in \text{dom}(\Gamma)$ 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 $\tau_2$.
- If $M$ is $\lambda x : \tau_2. M_1$, then $\tau$ is of the form $\tau_2 \to \tau_1$ 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 $\lambda$-calculus is *explicitly typed* (we also say in *church-style*), as parameters of abstractions are annotated with their types.

Simply-typed $\lambda$-calculus can also be *implicitly typed* (we also say in *curry-style*) when parameters of abstractions are left unannotated, as in the pure $\lambda$-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 \([M]\) for the type erasure of \(M\), which is defined by structural induction on \(M\):

\[
\begin{align*}
[x] & \triangleq x \\
[\lambda x : \tau . M] & \triangleq \lambda x . [M] \\
[M_1 M_2] & \triangleq [M_1] [M_2]
\end{align*}
\]
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

Annotating programs with types can lead to redundancy.

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

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

Full type reconstruction is undecidable for expressive type systems.

Some type annotations are required or type reconstruction is incomplete.
Untyped semantics

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

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

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

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

Obsiously, type erasure preserves reduction.

**Lemma (Direct simulation)**

If $M_1 \rightarrow M_2$ then $[M_1] \rightarrow [M_2]$.

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

**Lemma (Inverse simulation)**

If $[M] \rightarrow a$ then there exists $M'$ such that $M \rightarrow M'$ and $[M'] = a$.

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

- Simply-typed $\lambda$-calculus
- Type soundness for simply-typed $\lambda$-calculus
- Simple extensions: Pairs, sums, recursive functions
- Polymorphism
- Polymorphic $\lambda$-calculus
- Type soundness
- Type erasing semantics
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 $\tau$ in the empty environment.

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

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

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

The slogan now has a formal meaning:

Theorem (Type soundness)

Well-typed expressions do not go wrong.

Proof.
By Subject Reduction and Progress.

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

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

**Theorem (Subject reduction)**

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

**Theorem (Progress)**

*A (closed) well-typed term is either a value or reducible:*

*if $\emptyset \vdash M : \tau$ 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

- Simply-typed $\lambda$-calculus
- Type soundness for simply-typed $\lambda$-calculus
- Simple extensions: Pairs, sums, recursive functions
- Polymorphism
- Polymorphic $\lambda$-calculus
- Type soundness
- Type erasing semantics
Adding a unit

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

\[
M ::= \ldots | () \\
V ::= \ldots | ()
\]

No new reduction rule is introduced.

Types are extended with a new constant \textit{unit} and a new typing rule:

\[
\tau ::= \ldots | \text{unit} \\
\Gamma \vdash () : \text{unit}
\]
Pairs

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

Values, expressions, evaluation contexts are extended:

\[ M ::= \ldots | (M, M) | \text{proj}_i M \]
\[ E ::= \ldots | ([], M) | (V, []) | \text{proj}_i [] \]
\[ V ::= \ldots | (V, V) \]
\[ i \in \{1, 2\} \]

A new reduction rule is introduced:

\[ \text{proj}_i (V_1, V_2) \rightarrow V_i \]
Pairs

Types are extended:

\[ \tau ::= \ldots | \tau \times \tau \]

Two new typing rules are introduced:

\[
\text{PAIR} \quad \begin{array}{c}
\Gamma \vdash M_1 : \tau_1 \\
\Gamma \vdash M_2 : \tau_2 \\
\hline
\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2
\end{array}
\]

\[
\text{PROJ} \quad \begin{array}{c}
\Gamma \vdash M : \tau_1 \times \tau_2 \\
\Gamma \vdash \text{proj}_i M : \tau_i
\end{array}
\]
Sums

Values, expressions, evaluation contexts are extended:

\[
M ::= \ldots \mid inj_i M \mid case M of V \triangleright V \\
E ::= \ldots \mid inj_i [] \mid case [] of V \triangleright V \\
V ::= \ldots \mid inj_i V \\
i \in \{1, 2\}
\]

A new reduction rule is introduced:

\[
\text{case } inj_i V \text{ of } V_1 \triangleright V_2 \rightarrow V_i V
\]
Sums

Types are extended:

\[ \tau ::= \ldots \mid \tau + \tau \]

Two new typing rules are introduced:

\[ \text{INJ} \quad \quad \Gamma \vdash M : \tau_i \quad \quad \Gamma \vdash inj_i M : \tau_1 + \tau_2 \]

\[ \text{CASE} \quad \quad \Gamma \vdash M : \tau_1 + \tau_2 \quad \quad \Gamma \vdash V_1 : \tau_1 \rightarrow \tau \quad \quad \Gamma \vdash V_2 : \tau_2 \rightarrow \tau \]

\[ \Gamma \vdash \text{case } M \text{ of } V_1 \mid V_2 : \tau \]
Sums

with unique types

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

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

\[ V ::= \ldots | \text{inj}_i \; V \; \text{as} \; \tau \]

and modifying the typing rules and reduction rules accordingly.

Exercise

Describe an extension with the option type.
Modularity of extensions

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

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

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

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

These extensions are independent: they can be added to the $\lambda$-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 $\lambda$-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 $\lambda$-calculus is modified as follows.

Values and expressions are extended:

$$M ::= \ldots | \mu f : \tau. \lambda x. M$$
$$V ::= \ldots | \mu f : \tau. \lambda x. M$$

A new reduction rule is introduced:

$$(\mu f : \tau. \lambda x. M) \ V \longrightarrow [f \mapsto \mu f : \tau. \lambda x. M][x \mapsto V]M$$
Recursive functions

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

What does this imply as a corollary?

— Types will not distinguish functions from recursive functions.

A new typing rule is introduced:

\[ \text{FixAbs} \]

\[
\begin{array}{c}
\Gamma, f : \tau_1 \rightarrow \tau_2 \vdash \lambda x : \tau_1 . M : \tau_1 \rightarrow \tau_2 \\
\hline
\Gamma \vdash \mu f : \tau_1 \rightarrow \tau_2 . \lambda x . M : \tau_1 \rightarrow \tau_2
\end{array}
\]

In the premise, the type \( \tau_1 \rightarrow \tau_2 \) 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 \( \beta \)-redex "\((\lambda x : \tau. M_2) \) \( M_1 \)".

The latter can be type-checked \textit{only} by a derivation of the form:

\[
\begin{align*}
\text{APP} & \; \frac{\Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash (\lambda x : \tau_1. M_2) \ M_1 : \tau_2} \\
\text{ABS} & \; \frac{\Gamma \vdash \lambda x : \tau_1. M_2 : \tau_1 \to \tau_2 \quad \Gamma \vdash M_1 : \tau_1}{\Gamma, x : \tau_1 \vdash M_2 : \tau_2}
\end{align*}
\]

This means that the following \textit{derived rule} is sound and \textit{complete}:

\[
\begin{align*}
\text{LETMONO} & \; \frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \text{let} \ x : \tau_1 = M_1 \ \text{in} \ M_2 : \tau_2}
\end{align*}
\]

The construct "\( M_1 ; M_2 \)" can in turn be viewed as syntactic sugar for \textit{let} \( x : \text{unit} = M_1 \) \textit{in} \( M_2 \) where \( x \notin \text{ftv}(M_2) \).
A derived construct: let or a primitive one?

In the derived form \( \text{let } x : \tau_1 = M_1 \text{ in } M_2 \) 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 \( \text{let } x = M_1 \text{ in } M_2 \) with the following primitive typing rule.

\[
\text{LetMono} \\
\Gamma \vdash M_1 : \tau_1 \\
\Gamma, x : \tau_1 \vdash M_2 : \tau_2 \\
\_\_\_ \\
\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \tau_2
\]

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

Minimizing the number of language constructs is at least as important as avoiding extra type annotations in an explicitly-typed language.
A derived construct: let rec

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

\[
\frac{\Gamma, f : \tau \rightarrow \tau_1; x : \tau \vdash M_1 : \tau_1}{\Gamma \vdash \mu f : \tau \rightarrow \tau_1. \lambda x. M_1 : \tau \rightarrow \tau_1}
\]
\[
\frac{\Gamma, f : \tau \rightarrow \tau_1 \vdash M_2 : \tau_2}{\Gamma, f : \tau \rightarrow \tau_1 \vdash \mu f : \tau \rightarrow \tau_2. \lambda x. M_1 \ in \ M_2 : \tau_2}
\]

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

\[
\frac{\Gamma, f : \tau \rightarrow \tau_1; x : \tau \vdash M_1 : \tau_1}{\Gamma \vdash \text{let rec } (f : \tau \rightarrow \tau_1) \ x = M_1 \ in \ M_2 : \tau_2}
\]
Contents

- Simply-typed $\lambda$-calculus
- Type soundness for simply-typed $\lambda$-calculus
- Simple extensions: Pairs, sums, recursive functions

- Polymorphism

- Polymorphic $\lambda$-calculus
- Type soundness
- Type erasing semantics
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:

\[ \forall \alpha. (\alpha \to \alpha \to bool) \to list \alpha \to list \alpha \]

which *instantiates* to the monomorphic types:

\[ (int \to int \to bool) \to list int \to list int \]
\[ (bool \to bool \to bool) \to list bool \to list bool \]
\[ ... \]
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:

\[(T \rightarrow T \rightarrow bool) \rightarrow list T \rightarrow list T\]

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

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

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

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

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

By saying that this term admits the polymorphic type:

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

we make polymorphism internal to the type system.
Towards type abstraction

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

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

\[
\forall \alpha. (\alpha \to \alpha \to \text{bool}) \to \text{list } \alpha \to \text{list } \alpha
\]

then it *knows nothing* about \(\alpha\)—it is *parametric* in \(\alpha\)—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 \(\alpha\) is an *abstract type*. 

Parametricity

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

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

Similarly, the type of the list sorting function

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

reveals a "free theorem" about its behavior!

Basically, sorting commutes with $(\text{map } f)$, provided $f$ is order-preserving.

$$(\forall x, y, \text{cmp } (f x) (f y) = \text{cmp } x y) \implies$$

$$\forall \ell, \text{sort } (\text{map } f \ell) = \text{map } f (\text{sort } \ell)$$

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

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

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

Similarly, the type of the list sorting function

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

reveals a “free theorem” about its behavior!

Basically, sorting commutes with $(\text{map } f)$, provided $f$ is order-preserving.

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

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

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

There are two different (and sometimes incompatible) ways of defining this distinction...
Ad hoc v.s. parametric polymorphism: first definition

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

\[
\begin{align*}
\text{int} & \rightarrow \text{int}, \\
\text{bool} & \rightarrow \text{bool}, \\
\ldots \\
\forall \alpha. \alpha & \rightarrow \alpha
\end{align*}
\]

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

\[
\begin{align*}
\text{int} & \rightarrow \text{int} \rightarrow \text{int}, \\
\text{string} & \rightarrow \text{string} \rightarrow \text{string}, \\
\ldots \\
\text{but not} \\
\forall \alpha. \alpha & \rightarrow \alpha \rightarrow \alpha
\end{align*}
\]
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. $\lambda x. \text{show} \ (\text{read} \ x)$).
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:

$$\forall \alpha. \textit{Num} \alpha \Rightarrow \alpha \to \alpha \to \alpha$$

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

- Simply-typed λ-calculus
- Type soundness for simply-typed λ-calculus
- Simple extensions: Pairs, sums, recursive functions
- Polymorphism
  - Polymorphic λ-calculus
- Type soundness
- Type erasing semantics
System F

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

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

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

How are the syntax and semantics of terms extended?

There are several variants, depending on whether one adopts an

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

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

\[
\text{TABS} \quad \frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda \alpha. M : \forall \alpha. \tau} \quad \text{TAPP} \quad \frac{\Gamma \vdash M : \forall \alpha. \tau}{\Gamma \vdash M \tau' : [\alpha \mapsto \tau'] \tau}
\]

Terms are extended accordingly:

\[ M ::= \ldots | \Lambda \alpha. M | M \tau \]

Type variables are explicitly bound and appear in type environments.

\[ \Gamma ::= \ldots | \Gamma, \alpha \]
Well-formedness of environment

**Mandatory:** We extend our previous convention to form environments: \( \Gamma, \alpha \) requires \( \alpha \not\in \Gamma \), i.e. \( \alpha \) is neither in the domain nor in the image of \( \Gamma \).

**Optional:** We also require that environments be closed with respect to type variables, that is, we require \( \text{ftv}(\tau) \subseteq \text{dom}(\Gamma) \) 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:

- **WfEnv**
  - **WfEnv-EMPTY**
    - $\vdash \emptyset$
  - **WfEnvTvar**
    - $\vdash \Gamma \quad \alpha \notin \text{dom}(\Gamma)$
    - $\vdash \Gamma, \alpha$
  - **WfEnvVar**
    - $\Gamma \vdash \tau \quad x \notin \text{dom}(\Gamma)$
    - $\vdash \Gamma, x : \tau$

- **WfType**
  - **WfTypeVar**
    - $\vdash \Gamma \quad \alpha \in \Gamma$
    - $\vdash \Gamma \vdash \alpha$
  - **WfTypeArrow**
    - $\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2$
    - $\Gamma \vdash \tau_1 \to \tau_2$
  - **WfTypeForall**
    - $\Gamma, \alpha \vdash \tau$
    - $\Gamma \vdash \forall \alpha. \tau$

**Note**

Rule **WfEnvVar** need not the premise $\vdash \Gamma$, which follows from $\Gamma \vdash \tau$. 

63 412
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.

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

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:

\[(\Lambda \alpha. M) \tau \rightarrow [\alpha \mapsto \tau] M\]

Then, there is a choice.

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

\[V ::= \ldots \mid \Lambda \alpha. M\quad E ::= \ldots \mid [] \tau\]

However, this defines a type-passing semantics!

Indeed, \(\Lambda \alpha. ((\lambda y : \alpha. y) V)\) is then a value while its type erasure \((\lambda y. y) [V]\) is not—and can be further reduced.
Type-erasing semantics

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

\[ V ::= \ldots | \Lambda \alpha. V \quad\quad E ::= \ldots | \[] \tau | \Lambda \alpha. [] \]

Then, we only need a weaker version of \( \iota \)-reduction:

\[ (\Lambda \alpha. V) \tau \rightarrow [\alpha \mapsto \tau]V \quad (\iota) \]

We now have:

\[ \Lambda \alpha. ((\lambda y : \alpha. y) V) \rightarrow \Lambda \alpha. V \]

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

  \[ \text{if } f \text{ is list flattening of type } \forall \alpha. \text{list (list } \alpha) \to \text{list } \alpha, \text{ the monomorphic function } (f \text{ int}) \circ (f \text{ (list int)}) \text{ reduces to } \Lambda \alpha. f(\alpha) \circ (f \text{ (list } \alpha)), \text{ while its more general polymorphic version } \Lambda \alpha. (f \alpha) \circ (f \text{ (list } \alpha)) \text{ is irreducible.} \]

- 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)**

**Syntax**

\[ \tau \ ::= \alpha \mid \tau \to \tau \mid \forall \alpha. \tau \]

\[ M \ ::= x \mid \lambda x : \tau. M \mid M \ M \mid \Lambda \alpha. M \mid M \ \tau \]

**Typing rules**

**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**

\[ \frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1 \ M_2 : \tau_2} \]

**Tapp**

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

**Semantics**

\[ V \ ::= \lambda x : \tau. M \mid \Lambda \alpha. V \]

\[ E \ ::= [] M \mid V [] \mid [] \tau \mid \Lambda \alpha. [] \]

\[ (\lambda x : \tau. M) \ V \rightarrow [x \mapsto V]M \]

\[ (\Lambda \alpha. V) \ \tau \rightarrow [\alpha \mapsto \tau]V \]

**Context**

\[ M \rightarrow M' \]

\[ E[M] \rightarrow E[M'] \]
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:

\[
\text{pair} \triangleq \Lambda \alpha_1. \Lambda \alpha_2. \lambda x_1 : \alpha_1. \lambda x_2 : \alpha_2. \Lambda \beta. \lambda y : \alpha_1 \to \alpha_2 \to \beta. y \ x_1 \ x_2
\]

\[
\text{proj}_i \triangleq \Lambda \alpha_1. \Lambda \alpha_2. \lambda y : \forall \beta. (\alpha_1 \to \alpha_2 \to \beta) \to \beta. y \ \alpha_i \ (\lambda x_1 : \alpha_1. \lambda x_2 : \alpha_2. x_i)
\]

\[
[pair] \triangleq \lambda x_1. \lambda x_2. \lambda y. y \ x_1 \ x_2
\]

\[
[\text{proj}_i] \triangleq \lambda y. y \ (\lambda x_1. \lambda x_2. x_i)
\]

Sum and inductive types such as Natural numbers, List, etc. can also be encoded.
Primitive data-structures as constructors and destructors

Unit, Pairs, Sums, etc. can also be added to System F as primitives. We can then proceed as for simply-typed λ-calculus.

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

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

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

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

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:

$$Pair : \forall \alpha_1. \forall \alpha_2. \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

$$proj_i : \forall \alpha_1. \forall \alpha_2. \alpha_1 \times \alpha_2 \to \alpha_i$$

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

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

$$V ::= \ldots \mid Pair \mid Pair \tau \mid Pair \tau \tau \mid Pair \tau \tau V \mid Pair \tau \tau V V \mid proj_i \mid proj_i \tau \mid proj_i \tau \tau$$
Constructors and destructors

We add the two following reduction rules:

\[ \text{proj}_i \; \tau_1 \; \tau_2 \; (\text{pair} \; \tau'_1 \; \tau'_2 \; V_1 \; V_2) \rightarrow V_i \quad (\delta_{\text{pair}}) \]

Comments?

- For well-typed programs, \( \tau_i \) and \( \tau'_i \) will always be equal, but the reduction will not check this at runtime.

Instead, one could have defined the rule:

\[ \text{proj}_i \; \tau_1 \; \tau_2 \; (\text{pair} \; \tau_1 \; \tau_2 \; V_1 \; V_2) \rightarrow V_i \quad (\delta'_{\text{pair}}) \]

The two semantics are equivalent on well-typed terms, but differ on ill-typed terms where \( \delta'_{\text{pair}} \) may block when rule \( \delta_{\text{pair}} \) would progress, ignoring type errors.

Interestingly, with \( \delta'_{\text{pair}} \), the proof obligation is simpler for subject reduction but replaced by a stronger proof obligation for progress.
Constructors and destructors

We add the two following reduction rules:

\[ \text{proj}_i \, \tau_1 \, \tau_2 \, (\text{pair} \, \tau'_1 \, \tau'_2 \, V_1 \, V_2) \rightarrow V_i \quad (\delta_{\text{pair}}) \]

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

Assume given a collection of type constructors $G \in \mathcal{G}$, with their arity $\text{arity}(G)$. 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 $\tau$ is called a datatype if it is a $G$-type for some type constructor $G$.

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

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

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

We require that

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

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

\[
\begin{align*}
M & ::= \ldots | c \\
c & ::= C | d
\end{align*}
\]

\[
\frac{c : \tau \in \Delta}{\Gamma \vdash c : \tau} \quad \text{(CST)}
\]

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

\[
\begin{align*}
V & ::= \ldots \\
& | C \tau_1 \ldots \tau_p V_1 \ldots V_q \quad q \leq \text{arity} (C) \\
& | d \tau_1 \ldots \tau_p V_1 \ldots V_q \quad q < \text{arity} (d)
\end{align*}
\]

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

\[
d \tau_1 \ldots \tau_k V_1 \ldots V_n \rightarrow M \quad (\delta_d)
\]
Constructors and destructors

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

**Subject-reduction for constants:**

- $\delta$-rules preserve typings for well-typed terms

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

**Progress for constants:**

- Well-typed full applications of destructors can be reduced

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

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

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.
We introduce a destructor

\[ \text{fix} : \forall \alpha. \forall \beta. ((\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \in \Delta \]

of arity 2, together with the \( \delta \)-rule

\[ \text{fix} \ \tau_1 \ \tau_2 \ V_1 \ V_2 \longrightarrow \ V_1 \ (\text{fix} \ \tau_1 \ \tau_2 \ V_1) \ V_2 \quad (\delta_{\text{fix}}) \]

It is straightforward to check the assumptions:

- Progress is obvious, since \( \delta_{\text{fix}} \) works for any values \( V_1 \) and \( V_2 \).
- Subject reduction is also straightforward
  (by inspection of the typing derivation)

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

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

Solution

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

$$\forall \alpha \beta. \text{list} \alpha \rightarrow \beta \rightarrow (\alpha \rightarrow \text{list} \alpha \rightarrow \beta) \rightarrow \beta$$

with the two reduction rules:

$$\text{matchlist} \; \tau_1 \; \tau_2 \; (\text{Nil} \; \tau) \; V_n \; V_c \rightarrow V_n$$

$$\text{matchlist} \; \tau_1 \; \tau_2 \; (\text{Cons} \; \tau \; V_h \; V_t) \; V_n \; V_c \rightarrow V_c \; V_h \; V_t$$

2) See the case of pairs in the course.
Contents

- Simply-typed $\lambda$-calculus
- Type soundness for simply-typed $\lambda$-calculus
- Simple extensions: Pairs, sums, recursive functions
- Polymorphism
- Polymorphic $\lambda$-calculus
- Type soundness
- Type erasing semantics
Type soundness

The structure of the proof is similar to the case of simply-typed $\lambda$-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 $x \in \text{dom}(\Gamma)$ and $\Gamma(x) = \tau$.
- If $M$ is $\lambda x : \tau_0. M_1$, then $\tau$ is of the form $\tau_0 \rightarrow \tau_1$ and $\Gamma, x : \tau_0 \vdash M_1 : \tau_1$.
- If $M$ is $M_1 M_2$, then $\Gamma \vdash M_1 : \tau_2 \rightarrow \tau$ and $\Gamma \vdash M_2 : \tau_2$ for some type $\tau_2$.
- If $M$ is a constant $c$, then $c \in \text{dom}(\Delta)$ and $\Delta(x) = \tau$.
- If $M$ is $M_1 \tau_2$ then $\tau$ 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 $\tau$ 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

Lemma (Weakening)

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

1) If $x \not\in \Gamma$ and $\Gamma \vdash \tau'$, then $\Gamma, x : \tau' \vdash M : \tau$

2) If $\beta \not\in \Gamma$, then $\Gamma, \beta \vdash M : \tau$.

That is, if $\Gamma \vdash \Gamma', \Gamma''$, 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 $\Gamma_1 \not\# \Gamma_2$ then $\Gamma, \Gamma_2, \Gamma_1, \Gamma' \vdash M : \tau$. 
Type soundness

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 \( \Gamma' \). The proof is then straightforward—using the weakening lemma at variables.
Type soundness

Lemma (Type substitution, 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 $\Gamma'$ on the right. Then, the proof is straightforward.
Compositionality

Lemma (Compositionality)

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

Remarks

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

Theorem (Subject reduction)

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

The proof is by induction on $M$.

Using the previous lemmas it is straightforward.

Interestingly, the case for $\delta$-rules follows from the subject-reduction assumption for constants (slide 78).
Type soundness

Progress is restated as follows:

**Theorem (Progress, strengthened)**

A well-typed, irreducible closed term is a value:

if \( \vec{\alpha} \vdash M : \tau \) and \( M \rightarrow \), then \( M \) is some value \( V \).

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

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

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

Beware! We must take care of partial applications of constants

Lemma (Classification)

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

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

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

If $\tau$ is a $G$-constructed type (e.g. int, $\tau_1 \times \tau_2$, or $\tau\text{ list}$), 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

- Simply-typed $\lambda$-calculus
- Type soundness for simply-typed $\lambda$-calculus
- Simple extensions: Pairs, sums, recursive functions
- Polymorphism
- Polymorphic $\lambda$-calculus
- Type soundness
- Type erasing semantics
Implicitly-typed System F

The syntax and dynamic semantics of terms are that of the untyped $\lambda$-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 implicitly-typed simply-typed $\lambda$-calculus with two new rules:

\[
\begin{align*}
\text{IF-TABS} & \quad \frac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall \alpha.\tau} \\
\text{IF-TAPP} & \quad \frac{\Gamma \vdash a : \forall \alpha.\tau}{\Gamma \vdash a : [\alpha \mapsto \tau_0]\tau}
\end{align*}
\]

Notice that these rules are not syntax directed.
Implicitly-typed System F

On the side condition $\alpha \not\in \Gamma$

Notice that the explicit introduction of variable $\alpha$ in the premise of Rule TABS contains an implicit side condition $\alpha \not\in \Gamma$ due to the global assumption on the formation of $\Gamma, \alpha$:

$\text{IF-TABS}$

$$\frac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall \alpha.\tau}$$

$\text{IF-TABS-Bis}$

$$\frac{\Gamma \vdash a : \tau}{\Gamma \vdash a : \forall \alpha.\tau}$$

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

Then, we would need an explicit side-condition as in $\text{IF-TABS-Bis}$:

The side condition is important to avoid unsoundness by violation of the scoping rules.
Implicitly-typed System F

On the side condition $\alpha \not\in \Gamma$

Omitting the side condition leads to **unsoundness**:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VAR</strong></td>
<td>$x : \alpha_1 \vdash x : \alpha_1$</td>
</tr>
<tr>
<td><strong>Broken Tabs</strong></td>
<td>$\emptyset, x : \alpha_1 \vdash x : \forall \alpha_1. \alpha_1$</td>
</tr>
<tr>
<td><strong>TAPP</strong></td>
<td>$\emptyset, x : \alpha_1 \vdash x : \alpha_2$</td>
</tr>
<tr>
<td><strong>ABS</strong></td>
<td>$\emptyset \vdash \lambda x. x : \alpha_1 \rightarrow \alpha_2$</td>
</tr>
<tr>
<td><strong>Tabs-Bis</strong></td>
<td>$\emptyset \vdash \lambda x. x : \forall \alpha_1. \forall \alpha_2. \alpha_1 \rightarrow \alpha_2$</td>
</tr>
</tbody>
</table>

This is a type derivation for a **type cast** (Objective Caml’s Obj.magic).
Implicitly-typed System F

On the side condition $\alpha \not\in \Gamma$

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

- **Broken Var**
  \[
  \alpha_1, \alpha_2, x : \alpha_1, \alpha_1 \vdash x : \alpha_1
  \]

- **Broken Tabs**
  \[
  \alpha_1, \alpha_2, x : \alpha_1 \vdash x : \forall \alpha_1.\alpha_1
  \]

- **Tapp**
  \[
  \alpha_1, \alpha_2, x : \alpha_1 \vdash x : \alpha_2
  \]

- **Abs**
  \[
  \alpha_1, \alpha_2 \vdash \lambda x : \alpha_1 . x : \alpha_1 \rightarrow \alpha_2
  \]

- **Tabs**
  \[
  \emptyset \vdash \Lambda \alpha_1 . \Lambda \alpha_2 . \lambda \alpha_1 : x . x : \forall \alpha_1 . \forall \alpha_2 . \alpha_1 \rightarrow \alpha_2
  \]

\[\alpha_1, \alpha_2, x : \alpha_1, \alpha_1 \text{ ill-formed}\]
Implicitly-typed System F

On the side condition $\alpha \not\in \Gamma$

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

In that view, **Tabs-Bis** corresponds to the axiom:

$$\forall \alpha. (P \Rightarrow Q) \equiv P \Rightarrow (\forall \alpha. Q) \quad \text{if } \alpha \not\in P$$
Type-erasing typechecking

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

**Lemma**

Γ ⊢ a : τ holds in implicitly-typed System F if and only if there exists an explicitly-typed expression M whose erasure is a such that Γ ⊢ M : τ.

Trivial.

One could write judgements of the form Γ ⊢ a ⇒ M : τ to mean that the *explicitly typed* term M witnesses that the *implicitly typed* term a has type τ in the environment Γ.
An example

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

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

This term admits the polymorphic type:

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

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

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

This term admits the polymorphic type:

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

This begs the question: ...
Incomparable types in System F

Which of the two is more general?

\[ \forall \alpha_1. \forall \alpha_2. (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2 \]

\[ \forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2 \]

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

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

Take, for instance,

\[ \lambda f. \lambda x. \lambda y. (f \ y, f \ x) \]

and

\[ \lambda f. \lambda x. \lambda y. (f \ (f \ x), f \ (f \ y)) \]
Distrib pair in $F^\omega$ (parenthesis)

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

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

call it “dp” of type:

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

Then

$$dp (\lambda \alpha. \alpha)(\lambda \alpha. \alpha) : \forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

$$\Lambda \alpha_1. \Lambda \alpha_2. dp (\lambda \alpha. \alpha_1) (\lambda \alpha. \alpha_2) \alpha_1 \alpha_2 : \forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$
Notions of instance in $[F]$

It seems plausible that the untyped term $\lambda f x y. (f x, f y)$ does not admit a type $\tau_0$ of which the two previous types are instances.

But, in order to prove this, one must fix what it means for $\tau_2$ to be an instance of $\tau_1$—or, equivalently, for $\tau_1$ to be more general than $\tau_2$.

Several definitions are possible...
Syntactic notions of instance in $[F]$ 

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

\[
\text{Inst-Gen} \quad \frac{\vec{\beta} \neq \forall \vec{\alpha}.\tau}{\forall \vec{\alpha}.\tau \leq \forall \vec{\beta}.[\vec{\alpha} \mapsto \vec{\tau}]\tau}
\]

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

\[
\text{Sub} \quad \frac{\Gamma \vdash a : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash a : \tau_2}
\]

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

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

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

Well, $\tau_1$ and $\tau_2$ must be of the form $\forall \alpha. \tau$ and $\forall \bar{\beta}. [\bar{\alpha} \mapsto \bar{\tau}]\tau$ where $\bar{\beta} \# \forall \alpha. \tau$. W.l.o.g, we may assume that $\bar{\beta} \# \Gamma$.

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

\[
\begin{align*}
\text{Weak.} & \quad \frac{}{\Gamma \vdash M : \forall \alpha. \tau} \quad \bar{\beta} \# \Gamma \quad (1) \\
\text{Tapp}^* & \quad \frac{\Gamma, \bar{\beta} \vdash M : \forall \alpha. \tau}{\Gamma, \bar{\beta} \vdash M : [\alpha \mapsto \bar{\tau}]\tau} \\
\text{Tabs}^* & \quad \frac{\Gamma \vdash \Lambda \bar{\beta}. M : \forall \bar{\beta}. [\alpha \mapsto \bar{\tau}]\tau}{\Gamma \vdash \Lambda \bar{\beta}. M : \forall \bar{\beta}. [\bar{\alpha} \mapsto \bar{\tau}]\tau}
\end{align*}
\]

Admissible rule:

\[
\begin{align*}
\bar{\beta} \# \forall \alpha. \tau \quad (2) \\
\frac{\Gamma \vdash M : \forall \alpha. \tau}{\Gamma \vdash \Lambda \bar{\beta}. M : \forall \bar{\beta}. [\bar{\alpha} \mapsto \bar{\tau}]\tau}
\end{align*}
\]

If condition (2) holds, condition (1) may always be satisfied up to a renaming of $\bar{\beta}$. 
Retyping contexts in $F$

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

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

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

(Notice that $\mathcal{R}$ are arbitrarily deep, as opposed to evaluation contexts.) Let us write $\Gamma \vdash \mathcal{R}[\tau_1] : \tau_2$ iff $\Gamma, x : \tau_1 \vdash \mathcal{R}[x] : \tau_2$ (where $x \notin \mathcal{R}$)

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

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

In System $F$, retyping contexts can only change *toplevel* polymorphism: they cannot operate under arrow types to weaken the return type or strengthen the domain of functions.
Another syntactic notion of instance: $F_\eta$

Mitchell [1988] defined $F_\eta$, a version of $[F]$ extended with a richer instance relation as:

\[
\begin{align*}
\text{INST-GEN} \quad & \quad \hat{\beta} \not\equiv \forall \vec{\alpha}.\tau \\
\therefore \quad & \quad \forall \vec{\alpha}.\tau \leq \forall \hat{\beta}.[\vec{\alpha} \mapsto \vec{\tau}]\tau
\end{align*}
\]

\[
\text{DISTRIBUTIVITY} \quad \forall \alpha.(\tau_1 \to \tau_2) \leq (\forall \alpha.\tau_1) \to (\forall \alpha.\tau_2)
\]

\[
\begin{align*}
\text{CONGRUENCE-\rightarrow} \quad & \quad \tau_2 \leq \tau_1 \\
& \quad \tau_1' \leq \tau_2' \\
\therefore \quad & \quad \tau_1 \to \tau_1' \leq \tau_2 \to \tau_2'
\end{align*}
\]

\[
\begin{align*}
\text{CONGRUENCE-\forall} \quad & \quad \tau_1 \leq \tau_2 \\
\therefore \quad & \quad \forall \alpha.\tau_1 \leq \forall \alpha.\tau_2
\end{align*}
\]

\[
\begin{align*}
\text{TRANSITIVITY} \quad & \quad \tau_1 \leq \tau_2 \\
& \quad \tau_2 \leq \tau_3 \\
\therefore \quad & \quad \tau_1 \leq \tau_3
\end{align*}
\]

In $F_\eta$, Rule $\text{SUB}$ must be primitive as it is not admissible (but still sound).

$F_\eta$ can also be defined as the closure of System F under $\eta$-equality.

Why is a rich notion of instance potentially interesting?

- More polymorphism.
- More hope of having principal types.
A definition of principal typings

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

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

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

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

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

*A typing $\theta_1$ is more general than a typing $\theta_2$ if and only if every term that admits $\theta_1$ admits $\theta_2$ as well.*

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

This definition can be used to prove that a system does not have principal typings, under *any* reasonable definition of “instance”. 
Which systems have principal typings?

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

Wells [2002] shows that *neither System F nor $F_\eta$ have principal typings*.

It was shown earlier that *$F_\eta$’s instance relation is undecidable* [Wells, 1995; Tiuryn and Urzyczyn, 2002] and that *type inference for both System F and $F_\eta$ is undecidable* [Wells, 1999].
Which systems have principal typings?

There are still a few positive results...

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

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

Damas and Milner’s type system (coming up next) does not have *principal typings* but it has *principal types* and *decidable type inference*. 
Other approaches to type inference in System F

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

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

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

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 $[F]$?

*Progress?* Given a well-typed term $a \in [F]$, 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 $a_1 \in [F]$ of type $\tau$ 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 $\lambda$-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 $\beta$ or $\delta$ rules that are preserved by type-erasure, and $\iota$-steps corresponding to the reduction of type applications that disappear during type-erasure:
Type erasing semantics

Type erasure simulates in $[F]$ the reduction in $F$ upto $\iota$-steps:

**Lemma (Direct simulation)**

Assume $\Gamma \vdash M_1 : \tau$.

1. If $M_1 \xrightarrow{\iota} M_2$, then $[M_1] = [M_2]$.
2. If $M_1 \xrightarrow{\beta\delta} M_2$, then $[M_1] \xrightarrow{\beta\delta} [M_2]$.

Both parts are easy by definition of type erasure.
Type erasing semantics

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 $\lceil M_1 \rceil \rightarrow a$.

Then, there exists a term $M_2$ such that $M_1 \rightarrow^* \beta \delta M_2$ and $\lceil M_2 \rceil = a$. 
Type erasing semantics

Assumption on $\delta$-reduction

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

We first need $\delta$-reduction to be defined on type erasures.

- We may prove the theorem directly for some concrete examples of $\delta$-reduction.
  However, keeping $\delta$-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 $\delta$-reduction taken alone.
Type erasing semantics

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 $M \rightarrow_\delta M'$, then $\lceil M \rceil \rightarrow_\delta \lceil M' \rceil$.
2. If $\lceil M \rceil \rightarrow_\delta a$, then there exists $M'$ such that $M \rightarrow_\delta M'$ and $a$ is the type-erasuer of $M'$.

Remarks

- In most cases, the assumption on $\delta$-reduction is obvious to check.
- In general the $\delta$-reduction on untyped terms is larger than the projection of $\delta$-reduction on typed terms.
- If we restrict $\delta$-reduction to implicitly-typed terms, then it usually coincides with the projection of $\delta$-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 $[F]$ have a well-typed antecedent in $\iota$-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 $a_1 \rightarrow a_2$.

- 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 $M_1 \xrightarrow{\iota}^* \beta\delta M_2$ 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

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

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

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

The inverse simulation can first be shown assuming that $M_1$ is $\nu$-normal. The general case follows, since then $M_1$ $\nu$-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 $\nu$-reduction alone.

The termination of $\nu$-reduction is easy for System $F$, since it strictly decreases the number of type abstractions. (In $F^\omega$, it requires termination of simply-typed $\lambda$-calculus.)

The proof of inverse simulation in the case $M$ is $\nu$-normal is by induction on the reduction in $[\bar{F}]$, using a few helper lemmas, to deal with the fact that type-erasure is not injective.
Proof of inverse simulation

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

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

(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 \( [\bar{E}] \) is \( \bar{e} \) and \( [M] \) 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 \( \iota \)-normal form, then \( \mathcal{R} \) is of the form \( \Lambda \bar{\alpha}. [\!] \bar{\tau} \).
Proof of inverse simulation

Helper lemmas

Lemma (inversion of type erasure)

Assume $[M] = a$

- 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 $[M_1] = a_1$
- If $a$ is $a_1 \ a_2$, then $M$ is of the form $\mathcal{R}[M_1 \ M_2]$ with $[M_i] = a_i$

The proof is by induction on $M$. 
Lemma (Inversion of type erasure for well-typed values)

Assume $\Gamma \vdash M : \tau$ and $M$ is $\iota$-normal. If $[M]$ is a value $v$, then $M$ is a value $V$.

Moreover,

- If $v$ is $\lambda x. a_1$, then $V$ is $\Lambda \tilde{\alpha}. \lambda x: \tau. M_1$ with $[M_1] = a_1$.

- If $v$ is a partial application $c \, v_1 \ldots v_n$ then $V$ is $R[c \, \tilde{\tau} \, V_1 \ldots V_n]$ with $[V_i] = v_i$.

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 $\iota$-normal form whose erasure is $a$.

- If $a$ is $(\lambda x. a_1) \, v$, then $M$ is of the form $R[(\lambda x: \tau. M_1) \, V]$, with $[M_1] = a_1$ and $[V] = v$.

- If $a$ is a full application $(d \, v_1 \ldots v_n)$, then $M$ is of the form $R[d \, \tilde{\tau} \, V_1 \ldots V_n]$ and $[V_i]$ is $v_i$. 

Abstract Data types, Existential types, GADTs
Contents

- **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
Algebraic Datatypes Types

In OCaml:

```ocaml
type 'a list =
 | Nil : 'a list
 | Cons : 'a * 'a list -> 'a list
```

or

```ocaml
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

type $G \bar{\alpha} = \sum_{i \in 1..n}(C_i : \forall \bar{\alpha}. \tau_i \rightarrow G \bar{\alpha})$ where $\bar{\alpha} = \bigcup_{i \in 1..n} \text{ftv}(\tau_i)$

In System F, this amounts to declaring:

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

Exercise

*Show that this extension verifies the subject reduction and progress axioms for constants.*
Algebraic Datatypes Types

General case

\[
\text{type } G \overline{\alpha} = \Sigma_{i \in 1..n} (C_i : \forall \overline{\alpha}. \tau_i \to G \overline{\alpha}) \quad \text{where } \overline{\alpha} = \bigcup_{i \in 1..n} \text{ftv}(\tau_i)
\]

Notice that

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

Algebraic datatypes introduce \textit{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 $\tau ::= unit | \tau \times \tau | \tau + \tau$, 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:

\[ \text{list } \alpha \diamond \text{unit } + \alpha \times \text{list } \alpha \]

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:

$$\tau ::= \alpha \mid F \tilde{\tau} \mid \forall \beta. \tau$$

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

**Finite syntax for recursive types**

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

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 $\tau$ is *contractive*, that is, $\tau$ is a type constructor application or a forall introduction.

For instance, the type of lists of elements of type $\alpha$ is:

$$\mu\beta.(\text{unit} + \alpha \times \beta)$$
Equirecursive types

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

- **Fold/Unfold**
  \[ \mu \alpha \cdot \tau = [\alpha \mapsto \mu \alpha \cdot \tau] \tau \]

- **Uniqueness**
  \[ \tau_1 = [\alpha \mapsto \tau_1] \tau \quad \tau_2 = [\alpha \mapsto \tau_2] \tau \]
  \[ \tau_1 = \tau_2 \]

In both rules, \( \tau \) must be contractive.

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

**Co-inductive definition**

\[ \alpha = \alpha \quad \mu \alpha \cdot F \tilde{\tau} = [\alpha \mapsto \mu \alpha \cdot F \tilde{\tau}'] \tilde{\tau}' \]
\[ \mu \alpha \cdot F \tilde{\tau} = \mu \alpha \cdot F \tilde{\tau}' \]
\[ [\alpha \mapsto \mu \alpha \cdot \forall \beta \cdot \tau] \tau = [\alpha \mapsto \mu \alpha \cdot \forall \beta \cdot \tau'] \tau' \]
\[ \mu \alpha \cdot \forall \beta \cdot \tau = \mu \alpha \cdot \forall \beta \cdot \tau' \]
Equirecursive types

In the absence of quantifiers

Each type in this syntax denotes a unique regular tree, sometimes known as its \textit{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 \textit{decide} whether two types are \textit{equal}, that is, have identical infinite unfoldings.

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

Exercise

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

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

By coinduction

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

$\vdots$

By unification

Equivalent classes, using *small terms*

| $u \sim A u_1 \land u_1 \sim A u \land v \sim A v_1 \land v_1 \sim A v_2 \land v_2 \sim A v$ | $u \sim v$ |
| $u \sim A u_1 \sim v \sim A v_1 \land u_1 \sim A u \land v_1 \sim A v_2 \land v_2 \sim A v$ | $u_1 \sim v_1$ |
| $u \sim v \sim A v_1 \land u_1 \sim A u \sim v_1 \sim A v_2 \land v_2 \sim A v$ | $u f A v_2 v_4 12$ |
Equirecursive types

In the presence of quantifiers

The situation is more subtle because of $\alpha$-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

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

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

$$\forall a_1. \mu \ell. a_1 \rightarrow \forall a_2. (a_2 \rightarrow \ell)$$
$$\forall a_1. \mu \ell. a_1 \rightarrow \forall b_2. (b_2 \rightarrow \ell) \quad \quad (\alpha)$$
$$= \forall a_1. \quad a_1 \rightarrow \forall b_2. (b_2 \rightarrow \mu \ell. a_1 \rightarrow \forall b_2. (b_2 \rightarrow \ell)) \quad \quad (\mu)$$
$$= \forall a_1. \quad a_1 \rightarrow \forall b_2. (b_2 \rightarrow \mu \ell. a_1 \rightarrow \forall c_2. (c_2 \rightarrow \ell)) \quad \quad (\alpha)$$

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

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

- \( F \bar{\tau}_1 = F \bar{\tau}_2 \) implies \( \bar{\tau}_1 = \bar{\tau}_2 \) (symbols are injective)—this is used in the proof of Subject Reduction.
- \( F_1 \bar{\tau}_1 = F_2 \bar{\tau}_2 \) implies \( F_1 = F_2 \)—this is used in the proof of Progress.

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

**Exercise**

*Prove type soundness for the simply-typed \( \lambda \)-calculus in Coq. Then, change the syntax of types from Inductive to CoInductive.*
Equirecursive types

That is no a surprise, but...

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

All terms of the untyped $\lambda$-calculus are typable!

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

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

- references, or
- exceptions
Equirecursive types in OCaml

OCaml has both isorecursive and equirecursive types.

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

Quiz: why so?
Isorecursive types

The folding/unfolding is witnessed by an explicit coercion.

*The uniqueness rule is often omitted* (hence, the equality relation is weaker)

Encoding isorecursive types with ADT

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

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

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

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

A record can be defined as

\[
\text{type } G \vec{\alpha} = \Pi_{i \in 1..n} (\ell_i : \tau_i)
\]

where \( \vec{\alpha} = \bigcup_{i \in 1..n} \text{ftv}(\tau_i) \)

Exercise

What are the corresponding declarations in System F?

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

Can a record also be used for defining recursive types?

Exercise

Show type soundness for records.
Deep pattern matching

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

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

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

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

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

Exercise

*Do the transformation manually for the function equal.*
ADTs

If all occurrences of $G$ in $\tau_i$ are $G\ \bar{\alpha}$ 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.)
Non-regular ADT’s do not have this restriction:

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

They usually need *polymorphic* recursion to be manipulated.

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

(They are also typically used with GADTs.)

Non-regular ADT can actually be encoded in \( F^\omega \).
Contents

- 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
Existential types

Examples

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

\[ \exists \alpha. (\alpha \rightarrow \tau) \times \alpha \]

Type of closures in the environment-passing variant:

\[ \llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \exists \alpha. ((\alpha \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket) \times \alpha \]

A possible encoding of objects:

\[ = \exists \rho. \mu \alpha. \Pi (\{(\alpha \times \tau_1) \rightarrow \tau'_1; \ldots; (\alpha \times \tau_n) \rightarrow \tau'_n \}; \rho) \]

\[ \rho \text{ describes the state} \]

\[ \alpha \text{ is the concrete type of the closure} \]

\[ \text{a tuple...} \]

\[ \ldots \text{ that begins with a record...} \]

\[ \ldots \text{ of method code pointers...} \]

\[ \ldots \text{and continues with the state} \]

\[ \text{(a tuple of unknown length)} \]
Existential types

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

Here is how the existential quantifier is introduced and eliminated:

**Pack**

\[
\Gamma \vdash M : [\alpha \mapsto \tau']\tau
\]

\[
\Gamma \vdash \text{pack}\ \tau', M \text{ as } \exists\alpha.\tau : \exists\alpha.\tau
\]

**Unpack**

\[
\Gamma \vdash M_1 : \exists\alpha.\tau_1
\]

\[
\Gamma, \alpha, x : \tau_1 \vdash M_2 : \tau_2
\]

\[
\alpha \not\# \tau_2
\]

\[
\Gamma \vdash \text{let } \alpha, x = \text{unpack} \ M_1 \text{ in } M_2 : \tau_2
\]

Anything wrong? The side condition \(\alpha \not\# \tau_2\) is mandatory here to ensure well-formedness of the conclusion.

The side condition may also be written \(\Gamma \vdash \tau_2\) which implies \(\alpha \not\# \tau_2\), given that the well-formedness of the last premise implies \(\alpha \not\in \text{dom}(\Gamma)\).

Note the *imperfect duality* between universals and existentials:

**TAbs**

\[
\Gamma, \alpha \vdash M : \tau
\]

\[
\Gamma \vdash \Lambda\alpha. \ M : \forall\alpha.\tau
\]

**TApp**

\[
\Gamma \vdash M : \forall\alpha.\tau
\]

\[
\Gamma \vdash M \ \tau' : [\alpha \mapsto \tau']\tau
\]
On existential elimination

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

\[
\Gamma, \alpha \vdash M : \exists \alpha.\tau \\
\Gamma, \alpha \vdash \text{unpack } M : \tau
\]

Informally, this could mean that, if \( M \) has type \( \tau \) for some unknown \( \alpha \), then it has type \( \tau \), where \( \alpha \) is “fresh”...

Why is this broken?

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

Replacing the premise \( \Gamma, \alpha \vdash M : \exists \alpha.\tau \) by the conjunction \( \Gamma \vdash M : \exists \alpha.\tau \) and \( \alpha \in \text{dom}(\Gamma) \) 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 $\alpha$.

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

$$
\text{UNPACK} \quad 
\frac{
\Gamma \vdash M_1 : \exists \alpha. \tau_1 \quad 
\Gamma, \alpha; x : \tau_1 \vdash M_2 : \tau_2 \quad \alpha \neq \tau_2
}{
\Gamma \vdash \text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2 : \tau_2
}
$$

The restriction $\alpha \neq \tau_2$ prevents writing “let $\alpha, x = \text{unpack } M_1 \text{ in } x$”, which would be equivalent to the unsound “unpack $M$” of the previous slide.

The fact that $\alpha$ is bound within $M_2$ forces it to be treated abstractly.

In fact, $M_2$ must be $???$ in $\alpha$. 
On existential elimination

In fact, $M_2$ must be *polymorphic* in $\alpha$: the second premise could be:

$\Gamma \vdash M_1 : \exists \alpha. \tau_1$

$\Gamma, \alpha, x : \tau_1 \vdash \Lambda \alpha. \lambda x : \tau_1. M_2 : \forall \alpha. \tau_1 \rightarrow \tau_2 \quad \alpha \not\# \tau_2$

$\Gamma \vdash \text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2 : \tau_2$

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

$\Gamma \vdash M_1 : \exists \alpha. \tau_1 \quad \Gamma \vdash N_2 : \forall \alpha. \tau_1 \rightarrow \tau_2 \quad \alpha \not\# \tau_2$

$\Gamma \vdash \text{unpack } M_1 \ N_2 : \tau_2$

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

$\text{unpack}_{\exists \alpha. \tau_1} : (\exists \alpha. \tau_1) \rightarrow (\forall \alpha. (\tau_1 \rightarrow \tau_2)) \rightarrow \tau_2 \quad \alpha \not\# \tau_2$

Thus,

$\text{unpack}_{\exists \alpha. \tau} : \forall \beta. ((\exists \alpha. \tau) \rightarrow (\forall \alpha. (\tau \rightarrow \beta))) \rightarrow \beta$

or, better

$\text{unpack}_{\exists \alpha. \tau} : (\exists \alpha. \tau) \rightarrow \forall \beta. ((\forall \alpha. (\tau \rightarrow \beta)) \rightarrow \beta)$

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


On existential introduction

\[
\text{Pack} \\
\Gamma \vdash M : [\alpha \mapsto \tau']\tau \\
\overline{\Gamma \vdash \text{pack } \tau', M \text{ as } \exists \alpha. \tau : \exists \alpha. \tau}
\]

Hence, “\text{pack}_{\exists \alpha. \tau}” can be viewed as a family \textit{constant} of types:

\[
\text{pack}_{\exists \alpha. \tau} : \ [\alpha \mapsto \tau']\tau \rightarrow \exists \alpha. \tau
\]

\textit{i.e.} of polymorphic types:

\[
\text{pack}_{\exists \alpha. \tau} : \ \forall \alpha. (\tau \rightarrow \exists \alpha. \tau)
\]
Existentials as constants

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

\[
\text{pack}_{\exists \alpha.\tau} : \forall \alpha. (\tau \to \exists \alpha.\tau)
\]

\[
\text{unpack}_{\exists \alpha.\tau} : \exists \alpha.\tau \to \forall \beta. ((\forall \alpha. (\tau \to \beta)) \to \beta)
\]

Read:

- for \textit{any} \( \alpha \), if you have a \( \tau \), then, for \textit{some} \( \alpha \), you have a \( \tau \);
- if, for \textit{some} \( \alpha \), you have a \( \tau \), then, (for any \( \beta \),) if you wish to obtain a \( \beta \) out of it, you must present a function which, for \textit{any} \( \alpha \), obtains a \( \beta \) out of a \( \tau \).

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 \textit{double negation}:  
\[
\llbracket \exists \alpha. \tau \rrbracket = \forall \beta. ((\forall \alpha. (\llbracket \tau \rrbracket \to \beta)) \to \beta) \quad \text{if } \beta \neq \tau
\]

The term translation is:  
\[
\llbracket \text{pack}_{\exists \alpha. \tau} \rrbracket : \forall \alpha. (\llbracket \tau \rrbracket \to \llbracket \exists \alpha. \tau \rrbracket) \\
= \Lambda \alpha. \lambda x: \llbracket \tau \rrbracket. \Lambda \beta. \lambda k: \forall \alpha. (\llbracket \tau \rrbracket \to \beta). k \alpha x
\]

\[
\llbracket \text{unpack}_{\exists \alpha. \tau} \rrbracket : \llbracket \exists \alpha. \tau \rrbracket \to \forall \beta. ((\forall \alpha. (\llbracket \tau \rrbracket \to \beta)) \to \beta) \\
= \lambda x: \llbracket \exists \alpha. \tau \rrbracket. x
\]

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

What is the computational content of this encoding?

A \textit{continuation-passing transform}.

This encoding is due to Reynolds [1983], although it has more ancient roots in logic.
The semantics of existential types

`pack_{\exists \alpha.\tau}` can be treated as a unary constructor, and `unpack_{\exists \alpha.\tau}` as a unary destructor. The $\delta$-reduction rule is:

$$
\text{unpack}_{\exists \alpha.\tau_0} \left( \text{pack}_{\exists \alpha.\tau} \tau' V \right) \rightarrow \Lambda \beta. \lambda y : \forall \alpha.\tau \rightarrow \beta. y \; \tau' \; V
$$

It would be more intuitive, however, to treat `unpack_{\exists \alpha.\tau_0}` as a binary destructor:

$$
\text{unpack}_{\exists \alpha.\tau_0} \left( \text{pack}_{\exists \alpha.\tau} \tau' V \right) \tau_1 \left( \Lambda \alpha. \lambda x : \tau. M \right) \rightarrow [\alpha \mapsto \tau'] [x \mapsto V] M
$$

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

We extend values and evaluation contexts as follows:

\[
V ::= \ldots \mid \text{pack } \tau', V \text{ as } \tau \\
E ::= \ldots \mid \text{pack } \tau', [] \text{ as } \tau \mid \text{let } \alpha, x = \text{unpack } [] \text{ in } M
\]

We add the reduction rule:

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

Exercise

Show that subject reduction and progress hold.
The semantics of existential types

The reduction rule for existentials destructs its arguments.

Hence, \(\text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2\) 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 \(\text{let } x : \tau = M_1 \text{ in } M_2\) 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

Exercise

*Find an example that illustrates why the reduction of
let $\alpha, x = \text{unpack } M_1 \text{ in } M_2$ could be problematic when $M_1$ is not a value.*

*Need a hint?*

Use a conditional *Solution*

Let $M_1$ be if $M$ then $V_1$ else $V_2$ where $V_i$ is of the form

pack $\tau_i, W_i$ as $\exists \alpha. \tau$ and the two witnesses $\tau_1$ and $\tau_2$ 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:

\[
\Gamma \vdash M : [\alpha \mapsto \tau']\tau
\]

\[
\Gamma \vdash \text{pack } \tau', M \text{ as } \exists \alpha. \tau : \exists \alpha. \tau
\]

Isn't the witness type \(\tau'\) annotation superfluous?

- The type \(\tau_0\) of \(M\) is fully determined by \(M\). Given the type \(\exists \alpha. \tau\) of the packed value, checking that \(\tau_0\) 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 \(\tau'\). 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:

\[
\text{Unpack} \quad \frac{\Gamma \vdash a_1 : \exists \alpha. \tau_1 \quad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \quad \alpha \neq \tau_2}{\Gamma \vdash \text{let } x = a_1 \text{ in } a_2 : \tau_2}
\]

\[
\text{Pack} \quad \frac{\Gamma \vdash a : [\alpha \mapsto \tau'] \tau}{\Gamma \vdash a : \exists \alpha. \tau}
\]

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

The semantics of this let-binding is as before:

\[
E ::= \ldots | \text{let } x = E \text{ in } M \quad \text{let } x = V \text{ in } M \rightarrow [x \mapsto V]M
\]

Is the semantics type-erasing?
Implicitly-typed existential types

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:

\[
\text{let } x = M_1 \text{ in } M_2 \rightarrow [x \mapsto M_1]M_2
\]

With existential types, this breaks subject reduction!

Why?
Implicitly-typed existential types

Let $\tau_0$ be $\exists \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha)$ and $v_0$ a value of type $\text{bool}$. Let $v_1$ and $v_2$ be two values of type $\tau_0$ with incompatible witness types, e.g. $\lambda f. \lambda x. 1 + (f (1 + x))$ and $\lambda f. \lambda x. \text{not} (f (\text{not} x))$.

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

\[
\begin{align*}
a_1 &= \text{let } x = v v_0 \text{ in } x (x (\lambda y. y)) \\
\end{align*}
\]

We have $\emptyset \vdash a_1 : \exists \alpha. \alpha \to \alpha$ while $\emptyset \not\vdash a_2 : \tau$.

What happened? The term $a_1$ is well-typed since $v v_0$ has type $\tau_0$, hence $x$ can be assumed of type $(\beta \to \beta) \to (\beta \to \beta)$ for some unknown type $\beta$ 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 $\alpha$. This is demanding too much on its argument and the outer application is ill-typed.
Implicitly-typed existential types

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:

\[ \Gamma \vdash a_1 : \exists \alpha. \tau_1 \quad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \quad \alpha \not\equiv \tau_2 \]

\[ \Gamma \vdash [x \mapsto a_1]a_2 : \tau_2 \]

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

Existential types are trickier than they may appear at first.

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

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

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):

\[
\begin{align*}
\llbracket \text{unpack } a_1 (\lambda x. a_2) \rrbracket &= \llbracket a_1 \rrbracket (\lambda x. \llbracket a_2 \rrbracket) \quad & (1) \\
\rightarrow& (\lambda k. \llbracket a \rrbracket k) (\lambda x. \llbracket a_2 \rrbracket) \quad & (2) \\
\rightarrow& (\lambda x. \llbracket a_2 \rrbracket) \llbracket a \rrbracket \quad & (3) \\
\rightarrow& [x \mapsto \llbracket a \rrbracket][a_2] \quad & (4)
\end{align*}
\]

In the call-by-value setting, \( \lambda k. \llbracket a \rrbracket k \) would come from the reduction of \( \llbracket \text{pack } a \rrbracket \), i.e. is \( (\lambda k. \lambda x. k x) \llbracket a \rrbracket \), 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 \( \lambda k. \llbracket a \rrbracket k \).
• 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*:

\[ D \bar{\alpha} \approx \exists \bar{\beta}. \tau \quad \text{if} \quad \text{ftv}(\tau) \subseteq \bar{\alpha} \cup \bar{\beta} \quad \text{and} \quad \bar{\alpha} \neq \bar{\beta} \]

This introduces two constants, with the following type schemes:

\[
\begin{align*}
\text{pack}_D & : \forall \bar{\alpha} \bar{\beta}. \tau \to D \bar{\alpha} \\
\text{unpack}_D & : \forall \bar{\alpha} \gamma. D \bar{\alpha} \to (\forall \bar{\beta}. (\tau \to \gamma)) \to \gamma
\end{align*}
\]

(Compare with basic isorecursive types, where \( \bar{\beta} = \emptyset \).)
Iso-existential types in ML

One point has been hidden on the previous slide. The “type scheme:"
\[
\forall \bar{\alpha} \gamma. D \bar{\alpha} \rightarrow (\forall \bar{\beta}. (\tau \rightarrow \gamma)) \rightarrow \gamma
\]
is in fact *not* an ML type scheme. How could we address this?

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

\[
\text{UNPACK}_D
\]

\begin{align*}
\Gamma \vdash M_1 : D \bar{\tau} \\
\Gamma \vdash M_2 : \forall \bar{\beta}. ([\bar{\alpha} \mapsto \bar{\tau}] \tau \rightarrow \tau_2) \\
\bar{\beta} \not\# \bar{\tau}, \tau_2
\end{align*}

\[
\Gamma \vdash \text{unpack}_D M_1 M_2 : \tau_2
\]

where \(D \bar{\alpha} \approx \exists \bar{\beta}. \tau\)

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

The term \(M_2\) must be polymorphic, which \texttt{GEN} can prove.
Iso-existential types in ML

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

The constant \( \text{pack}_D \) admits an ML type scheme, so it is unproblematic.

The construct \( \text{unpack}_D \) leads to this constraint generation rule (see type inference):

\[
\langle \text{unpack}_D \ M_1 \ M_2 : \tau_2 \rangle = \exists \bar{\alpha}. \left( \langle M_1 : D \ \bar{\alpha} \rangle \ \forall \bar{\beta}. \langle M_2 : \tau \rightarrow \tau_2 \rangle \right)
\]

where \( D \ \bar{\alpha} \approx \exists \bar{\beta}. \tau \) and, \( \text{w.l.o.g.,} \ \bar{\alpha} \bar{\beta} \not\equiv M_1, M_2, \tau_2 \).

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:

\[
\text{type } D \vec{\alpha} = \ell : \tau \to D \vec{\alpha}
\]

where \(\ell\) is a data constructor and \(\vec{\beta}\) appears free in \(\tau\) but does not appear in \(\vec{\alpha}\). The elimination construct is typed as:

\[
\langle \text{match } M_1 \text{ with } \ell x \to M_2 : \tau_2 \rangle = \exists \vec{\alpha}. \left( \langle M_1 : D \vec{\alpha} \rangle \ \forall \vec{\beta}. \text{def } x : \tau \text{ in } \langle M_2 : \tau_2 \rangle \right)
\]

where, w.l.o.g., \(\vec{\alpha} \vec{\beta} \# M_1, M_2, \tau_2\).
An example

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

\[
\langle \text{unpack}_{\text{Any}} M_1 (\lambda x. x) : \tau_2 \rangle = \langle M_1 : \text{Any} \rangle \land \forall \beta. \langle \lambda x. x : \beta \to \tau_2 \rangle
\]
\[
\triangleright \quad \forall \beta. \beta = \tau_2
\]
\[
\equiv \quad \text{false}
\]

(Recall that \( \beta \not\approx \tau_2 \).)
An example

Define

\[ D \alpha \approx \exists \beta. (\beta \to \alpha) \times \beta \]

A client that regards \( \beta \) as abstract succeeds:

\[
\langle \text{unpack}_D \ M_1 \ (\lambda (f, y). f \ y) : \tau \rangle
\]
\[
= \exists \alpha. (\langle M_1 : D \alpha \rangle \land \forall \beta. \langle \lambda (f, y). f \ y : ((\beta \to \alpha) \times \beta) \to \tau \rangle)
\]
\[
\equiv \exists \alpha. (\langle M_1 : D \alpha \rangle \land \forall \beta. \text{def } f : \beta \to \alpha; y : \beta \text{ in } \langle f \ y : \tau \rangle)
\]
\[
\equiv \exists \alpha. (\langle M_1 : D \alpha \rangle \land \forall \beta. \tau = \alpha)
\]
\[
\equiv \exists \alpha. (\langle M_1 : D \alpha \rangle \land \tau = \alpha)
\]
\[
\equiv \langle M_1 : D \tau \rangle
\]
Existential types calls for universal types!

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

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

\[
\text{List.map } (\lambda (z) \text{ let } Delay (f, y) = z \text{ in } Delay ((\lambda (z) g (f z)), y))
\]

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

\[
\text{let lift } g \ l = \\
\text{List.map } (\lambda (z) \text{ let } Delay (f, y) = z \text{ in } Delay ((\lambda (z) g (f z)), y))
\]

In expression \( let \alpha, x = \text{unpack } M_1 \ in \ M_2 \), occurrences of \( x \) in \( M_2 \) can only be passed to external functions (free variables) that are polymorphic in \( \alpha \) so that \( \alpha \) 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 writing type annotations, but in a more rigid way
- In particular, there is no canonical way of representing them. For example, a thunk of type $\exists \beta (\beta \to \text{int}) \times \beta$ could have been defined as `Delay (succ, 1)` where `Delay` is either one of

```plaintext
type int_thunk = Delay : ('b \to \text{int}) \times \text{'b} -> int_thunk
type 'a thunk = Delay : ('b \to 'a) \times \text{'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:

\[
\exists \text{stack}. \{ \text{empty} : \text{stack}; \text{push} : \text{int} \times \text{stack} \rightarrow \text{stack}; \text{pop} : \text{stack} \rightarrow \text{option} (\text{int} \times \text{stack}) \}\]

specifies an abstract implementation of integer stacks.

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

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

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

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

Via GADT (iso-existential types)

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

Via first-class modules (abstract types)

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

- 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
An introduction to GADTs
What are they?

ADTs

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

\[
\text{type } \alpha \ \text{list} = \\
\quad \text{Nil } : \alpha \ \text{list} \\
\quad \text{Const } : \alpha \ast \alpha \ \text{list} \to \alpha \ \text{list}
\]

GADTs

This is no more the case with GADTs

\[
\text{type } (\alpha, \beta) \ \text{eq} = \\
\quad \text{Eq } : (\alpha, \alpha) \ \text{eq} \\
\quad \text{Any } : (\alpha, \beta) \ \text{eq}
\]

The \text{Eq} constructor may only build values of types of \((\alpha, \alpha)\) \text{eq}.
For example, it cannot build values of type \((\text{int, string})\) \text{eq}.

The criteria is \textit{per constructor}: it remains a GADT when another (even \textit{regular}) constructor is added.
Examples

1. Let add \((x, y) = x + y\) in
2. Let not \(x = \text{if } x \text{ then false else true}\) in
3. Let body \(b = \)
   - Let step \(x = \)
     - add \((x, \text{if not } b \text{ then } 1 \text{ else } 2)\)
   - In step (step 0))
   In body true

Define a single apply function that dispatches all function calls:

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

Defunctionalization

Introduce a constructor per function

```
type \((\_, \_\_\_)\) apply =
  | Fadd : (int \(\times\) int, int) apply
  | Fnot : (bool, bool) apply
  | Fbody : (bool, int) apply
  | Fstep : bool \(\to\) (int, int) apply
```
Examples

A typed abstract-syntax tree

```ocaml
type _ expr =
| Int    : int -> int expr
| Zerop  : int expr -> bool expr
| If     : (bool expr * α expr * α expr) -> α expr

let e0 : int expr = (If (Zerop (Int 0), Int 1, Int 2))
```

A typed evaluator (with no failure)

```ocaml
let rec eval : type a . a expr -> a =
    fun x ->
    match x with
    | Int x -> x
    | Zerop x -> eval x > 0
    | If (b, e1, e2) -> if eval b then eval e1 else eval e2

let b0 = eval e0
```

Exercise

What would you have to do without GADTs? Define a typed abstract syntax tree for the simply-typed \(\lambda\)-calculus and a typed evaluator.
Examples

Example of printing

define type ty =
  | Tint : int ty
  | Tbool : bool ty
  | Tlist : \alpha ty \rightarrow (\alpha list) ty
  | Tpair : \alpha ty \times \beta ty \rightarrow (\alpha \times \beta) ty

let rec to_string : type a. a ty \rightarrow a \rightarrow string = fun t x \rightarrow match t with
  | Tint \rightarrow string_of_int x
  | Tbool \rightarrow if x then "true" else "false"
  | Tlist t \rightarrow "[" ^ String.concat ";" (List.map (to_string t) x) ^ "]
  | Tpair (a, b) \rightarrow
    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 ∗ γ → (α, β) 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 ∗ γ ∗ bool → (α, β) prod

let get (type a b) (p : (a, b) prod) : bool =
  let Prod (t, v, s) = p in s
Examples

**Exercise**
Specialize the encoding of sum types to the encoding of 'a list
Other uses of GADTs

GADTs

- May encode data-structure invariants, such as the state of an automaton, as illustrated by Pottier and Régis-Gianas [2006] for typechecking LR-parsers.

- They may be used to implement a form of dynamic type (similarly to the generic printer)

- They may be used to optimize representation (e.g. sum’s encoding)

- GADTs can be used to encode type classes, using a technique analogous to defunctionalization [Pottier and Gauthier, 2006].
Reducing GADTs to type equality  
(and existential types)

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

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

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

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

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

▷ We enlarge the domain of each constructor,

▷ But require a proof evidence as an extra argument that a certain equality holds to restrict the possible uses of the constructors.

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

let s = to_string (Tpair (Eq, Tlist (Eq, Tint Eq), Tint Eq)) ([1; 2; 3], 0)
```
Reducing GADTs to type equality  
(and existential types)

All GADTs can be encoded with a single one:

\[
\text{type} \ (\alpha, \beta) \ eq = \ Eq : (\alpha, \alpha) \ eq
\]

For instance, generic programming can be redefined as follows:

\[
\text{type} \ \alpha \ ty =
\begin{align*}
| & \text{Tint} : (\alpha, \text{int}) \ eq \to \alpha \ ty \\
| & \text{Tlist} : (\alpha, \beta \ \text{list}) \ eq \ast \beta \ ty \to \alpha \ ty \\
| & \text{Tpair} : (\alpha, (\beta \ast \gamma)) \ eq \ast \beta \ ty \ast \gamma \ ty \to \alpha \ ty
\end{align*}
\]

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

\[
\text{let rec to_string : type} \ a. \ a \ ty \to \ a \to \text{string} = \ \text{fun} \ t \ x \to \text{match} \ t \ \text{with} \\
| & \text{Tint} \ Eq \to \text{string_of_int} \ x \\
| & \text{Tlist} \ (Eq, \ l) \to \text{...} \\
| & \text{Tpair} \ (Eq, \ a, \ b) \to \text{...}
\]

▷ Pattern “Tint Eq” is GADT matching

GADTs
Existential types
Algebraic Data Types
Typed closure conversion
Reducing GADTs to type equality (and existential types)

All GADTs can be encoded with a single one:

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

For instance, generic programming can be redefined as follows:

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

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

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

▷ Pattern “Tint Eq” is GADT matching
▷ let Eq = p in... introduces the equality a = int in the current branch
Formalisation of GADTs

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

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

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

When typechecking an expression

$$E[\ let \ x : \tau_1 \sim \tau_2 = M_0 \ in \ M] \quad E[\lambda x : \tau_1 \sim \tau_2. M]$$

- $M$ is typechecked with the assumption that $\tau_1 \sim \tau_2$, i.e. types $\tau_1$ and $\tau_2$ are equivalent, which allows for type conversion within $M$

- but $E$ and $M_0$ are typechecked without this assumption

- 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).
Add equality coercions to System $F$

Coercions witness type equivalences:

Types

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

Expressions

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

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

Typing rules:

Coerce

$$\Gamma \vdash M : \tau_1$$
$$\Gamma \vdash \gamma : \tau_1 \sim \tau_2$$
$$\Gamma \vdash \gamma \triangleright M : \tau_2$$

Coercion

$$\Gamma \vdash \gamma : \tau_1 \sim \tau_2$$
$$\Gamma \vdash \lambda x : \tau_1 \sim \tau_2. \ M : \tau_1 \sim \tau_2 \rightarrow \tau$$
## Fc (simplified)

### Typing of coercions

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EQ-HYP</strong></td>
<td>( y : \tau_1 \sim \tau_2 \in \Gamma )</td>
<td>( \Gamma \vdash y : \tau_1 \sim \tau_2 )</td>
</tr>
<tr>
<td><strong>EQ-REF</strong></td>
<td>( \Gamma \vdash \gamma : \tau )</td>
<td>( \Gamma \vdash \langle \tau \rangle : \tau \sim \tau )</td>
</tr>
<tr>
<td><strong>EQ-SYM</strong></td>
<td>( \Gamma \vdash \gamma : \tau_1 \sim \tau_2 )</td>
<td>( \Gamma \vdash \text{sym} \gamma : \tau_2 \sim \tau_1 )</td>
</tr>
<tr>
<td><strong>EQ-TRANS</strong></td>
<td>( \Gamma \vdash \gamma_1 : \tau_1 \sim \tau ) ( \Gamma \vdash \gamma_2 : \tau \sim \tau_2 )</td>
<td>( \Gamma \vdash \gamma_1 ; \gamma_2 : \tau_1 \sim \tau_2 )</td>
</tr>
<tr>
<td><strong>EQ-ARROW</strong></td>
<td>( \Gamma \vdash \gamma_1 : \tau_1 \sim \tau ) ( \Gamma \vdash \gamma_2 : \tau_2 \sim \tau_2' )</td>
<td>( \Gamma \vdash \gamma_1 \rightarrow \gamma_2 : \tau_1 \rightarrow \tau_2 \sim \tau_1' \rightarrow \tau_2' )</td>
</tr>
<tr>
<td><strong>EQ-LEFT</strong></td>
<td>( \Gamma \vdash \gamma : \tau_1 \rightarrow \tau_2 \sim \tau_1' \rightarrow \tau_2' )</td>
<td>( \Gamma \vdash \text{left} \gamma : \tau_1' \sim \tau_1 )</td>
</tr>
<tr>
<td><strong>EQ-RIGHT</strong></td>
<td>( \Gamma \vdash \gamma : \tau_1 \rightarrow \tau_2 \sim \tau_1' \rightarrow \tau_2' )</td>
<td>( \Gamma \vdash \text{right} \gamma : \tau_2 \sim \tau_2' )</td>
</tr>
<tr>
<td><strong>EQ-ALL</strong></td>
<td>( \Gamma, \alpha \vdash \gamma : \tau_1 \sim \tau_2 )</td>
<td>( \Gamma \vdash \forall \alpha. \gamma : \forall \alpha. \tau_1 \sim \forall \alpha. \tau_2 )</td>
</tr>
<tr>
<td><strong>EQ-INST</strong></td>
<td>( \Gamma \vdash \gamma : \forall \alpha. \tau_1 \sim \forall \alpha. \tau_2 ) ( \Gamma \vdash \tau )</td>
<td>( \Gamma \vdash \gamma@\tau : [\alpha \mapsto \tau] \tau_1 \sim [\alpha \mapsto \tau] \tau_2 )</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
(\gamma \triangleleft V_1) V_2 & \rightarrow \text{right } \gamma \triangleleft (V_1 \ (\text{left } \gamma \triangleleft V_2)) \\
(\gamma \triangleleft V) \tau & \rightarrow \ (\gamma@\tau) \triangleleft (V \ \tau) \\
\gamma_1 \triangleleft (\gamma_2 \triangleleft V) & \rightarrow \ (\gamma_1; \gamma_2) \triangleleft V
\end{align*}
\]
Semantics

Coercions should be without computational content

Except for coercion abstractions that must stop the evaluation

- Otherwise, one could attempt to reduce $M$ in $\lambda int \sim bool. M$ when $M$ is not $(bool \triangleleft 0)$, 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 performed, but be aware of stuck programs and treat them as dead branches.
Type soundness

By subject reduction and progress with explicit coercions

Erasing semantics

Important and not so obvious.

\[ \gamma \triangleleft M \text{ erases to } M \]
\[ \gamma \text{ erases to } \diamond \]

Slogan that “coercion have 0-bit information”, i.e.

Coercions need not be passed at runtime—but still block the reduction.

Expressions and typing rules.

\[ \text{Coerce} \]
\[ \Gamma \vdash M : \tau_1 \]
\[ \Gamma \vdash \diamond : \tau_1 \sim \tau_2 \]
\[ \Gamma \vdash M : \tau_2 \]

\[ \text{Coercion} \]
\[ \Gamma \vdash \tau_1 \sim \tau_2 \]
\[ \Gamma \vdash \diamond : \tau_1 \sim \tau_2 \]

\[ \text{Coabs} \]
\[ \Gamma, x : \tau_1 \sim \tau_2 \vdash M : \tau \]
\[ \Gamma \vdash \lambda x : \tau_1 \sim \tau_2. M : \tau_1 \sim \tau_2 \rightarrow \tau \]
Type soundness

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

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 outside 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

- 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
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 $\llbracket M \rrbracket$, but also a translation of *types*, mapping $\tau$ to $\llbracket \tau \rrbracket$, 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 \( \text{fv}(\lambda x. a) \):

\[
\begin{align*}
[\lambda x. a] &= \text{let } code = \lambda(clo, x). \\
&\quad \text{let } (\_, x_1, \ldots, x_n) = \text{clo in } [a] \text{ in} \\
&\quad (code, x_1, \ldots, x_n) \\
\end{align*}
\]

\[
\begin{align*}
[a_1 \ a_2] &= \text{let } clo = [a_1] \text{ in} \\
&\quad \text{let } code = \text{proj}_0 \clo \text{ in} \\
&\quad code (clo, [a_2]) \\
\end{align*}
\]

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

**Important!** The layout of the environment must be known only at the closure allocation site, not at the call site. In particular, \( \text{proj}_0 \clo \) need not know the size of \( clo \).
Environment-passing closure conversion

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

\[
\begin{align*}
\llbracket \lambda x. a \rrbracket &= \text{let code} = \lambda (env, x). \\
&\quad \text{let } (x_1, \ldots, x_n) = env \text{ in } \llbracket a \rrbracket \text{ in} \\
&\quad (\text{code}, (x_1, \ldots, x_n)) \\
\llbracket a_1 a_2 \rrbracket &= \text{let } (\text{code}, env) = \llbracket a_1 \rrbracket \text{ in} \\
&\quad \text{code} (env, \llbracket a_2 \rrbracket)
\end{align*}
\]

Questions: How can closure conversion be made \textit{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, \( \llbracket \tau_1 \to \tau_2 \rrbracket \)?
Environment-passing closure conversion

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

\[
\begin{align*}
[\lambda x. a] &= \begin{align*}
\text{let code} &= \lambda (\text{env}, x).
\text{let } (x_1, \ldots, x_n) &= \text{env in } [a] \text{ in }
(c \text{ode}, (x_1, \ldots, x_n))
\end{align*}
\end{align*}
\]

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

Write \([\Gamma]\) for the tuple type \( x_1 : [\tau'_1]; \ldots; x_n : [\tau'_n] \) where \( \Gamma \) 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

- \( \text{env} \) has type \([\Gamma]\),
- \( \text{code} \) has type \(([\Gamma] \times [\tau_1]) \rightarrow [\tau_2]\), and
- the entire closure has type \((([\Gamma] \times [\tau_1]) \rightarrow [\tau_2]) \times [\Gamma]\).

Now, \emph{what should be the definition of} \([\tau_1 \rightarrow \tau_2]\)?
Towards a type translation

Can we adopt this as a definition?

\[
[\tau_1 \rightarrow \tau_2] = \left( ([\Gamma] \times [\tau_1]) \rightarrow [\tau_2] \right) \times [\Gamma]
\]

Naturally not. This definition is mathematically ill-formed: we cannot use \( \Gamma \) out of the blue.

That is, this definition is not uniform: it depends on \( \Gamma \), \( i.e. \) the size and layout of the environment.

Do we really need to have a uniform translation of types?
Towards a type translation

Yes, we do.

*We need a uniform translation of types*, not just because it is nice to have one, but because it describes a *uniform calling convention*.

If closures with distinct environment sizes or layouts receive distinct types, then we will be unable to translate this well-typed code:

\[
\text{if } \ldots \text{ then } \lambda x. x + y \text{ else } \lambda x. x
\]

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:

\[
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \exists \alpha.((\alpha \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket) \times \alpha
\]

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 \( \alpha \) 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:

\[
let f = let x = ref 0 in \lambda().x := (x + 2);!x
\]

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:

$$\Gamma \vdash M : \tau \quad \text{implies} \quad \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket \tau \rrbracket$$
Environment-passing closure conversion

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

$$\llbracket \lambda x : \tau_1. M \rrbracket = \text{let code : } ([\Gamma] \times [\tau_1]) \to [\tau_2] = \lambda (env : [\Gamma], x : [\tau_1]).$$

$$\text{let } (x_1, \ldots, x_n : [\Gamma]) = env \text{ in } [M]$$

in

$$\text{pack } [\Gamma], (\text{code}, (x_1, \ldots, x_n))$$

as $\exists \alpha. ((\alpha \times [\tau_1]) \to [\tau_2]) \times \alpha$

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 $\Gamma \vdash M_1 : \tau_1$.

$\llbracket M \; M_1 \rrbracket = \text{let } \alpha, (\text{code} : (\alpha \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket, env : \alpha) = \text{unpack} \llbracket M \rrbracket \text{ in }$

$\text{code} (env, \llbracket M_1 \rrbracket)$

We find $\llbracket \Gamma \rrbracket \vdash \llbracket M \; M_1 \rrbracket : \llbracket \tau_2 \rrbracket$, as desired.
Recursive functions can be translated in this way, known as the “fix-code” variant [Morrisett and Harper, 1998] (leaving out type information):

\[
\llbracket \mu f.\lambda x.M \rrbracket = \text{let rec code (env, } x) =
\]

\[
\quad \text{let } f = \text{pack (code, env) in}
\]

\[
\quad \text{let } (x_1, \ldots, x_n) = \text{env in}
\]

\[
\llbracket M \rrbracket \text{ in}
\]

\[
\quad \text{pack (code, } (x_1, \ldots, x_n) \text{)}
\]

where \( \{x_1, \ldots, x_n\} = \text{fv}(\mu f.\lambda x.M) \).

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.
Instead, the “fix-pack” variant [Morrisett and Harper, 1998] uses an extra field in the environment to store a back pointer to the closure:

\[
\llbracket \mu f. \lambda x. M \rrbracket = \text{let code } (env, x) = \\
\text{let } (f, x_1, \ldots, x_n) = env \text{ in } \\
\llbracket M \rrbracket \\
in \\
\text{let rec } clo = (\text{code}, (clo, x_1, \ldots, x_n)) \text{ in } clo
\]

where \( \{x_1, \ldots, x_n\} = \text{fv}(\mu f. \lambda x. M) \).

This requires general, recursively-defined values. Closures are now cyclic data structures.
Here is how the “fix-pack” variant is type-checker. Assume
\( \Gamma \vdash \mu f. \lambda x. M : \tau_1 \rightarrow \tau_2 \) and \( \text{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \text{fv}(\mu f. \lambda x. M) \).

\[
\llbracket \mu f : \tau_1 \rightarrow \tau_2, \lambda x. M \rrbracket =
\]

let code : \( (\llbracket f : \tau_1 \rightarrow \tau_2; \Gamma \rrbracket \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket = \ Nation \ (\text{env} : \llbracket f : \tau_1 \rightarrow \tau_2, \Gamma \rrbracket, x : \llbracket \tau_1 \rrbracket).

let \( (f, x_1, \ldots, x_n) : \llbracket f : \tau_1 \rightarrow \tau_2, \Gamma \rrbracket = \text{env} \) in
\( \llbracket M \rrbracket \) in

let rec clo : \llbracket \tau_1 \rightarrow \tau_2 \rrbracket =
pack \( \llbracket f : \tau_1 \rightarrow \tau_2, \Gamma \rrbracket, (\text{code}, (clo, x_1, \ldots, x_n)) \)
as \( \exists \alpha((\alpha \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket) \times \alpha \)
in clo

Problem?
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 \textit{indirect} compilation of polymorphic recursive functions as long as the recursion is monomorphic.

Fortunately, the encoding can be straightforwardly adapted to \textit{directly} compile polymorphically recursive functions into polymorphic closure.
Environment-passing closure conversion

\[ \mu f : \forall \vec{\beta}. \tau_1 \to \tau_2. \lambda x. M \] =

\[ \text{let code} : \forall \vec{\beta}. (\llbracket f : \forall \vec{\beta}. \tau_1 \to \tau_2; \Gamma \rrbracket \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket = \]

\[ \lambda (env : \llbracket f : \forall \vec{\beta}. \tau_1 \to \tau_2, \Gamma \rrbracket, x : \llbracket \tau_1 \rrbracket).\]

\[ \text{let} \ (f, x_1, \ldots, x_n) : \llbracket f : \forall \vec{\beta}. \tau_1 \to \tau_2, \Gamma \rrbracket = env \text{ in} \]

\[\llbracket M \rrbracket \text{ in} \]

\[\text{let rec clo} : \llbracket \forall \vec{\beta}. \tau_1 \to \tau_2 \rrbracket = \]

\[ \Lambda \vec{\beta}. \text{pack} \llbracket f : \forall \vec{\beta}. \tau_1 \to \tau_2, \Gamma \rrbracket, (\text{code } \vec{\beta}, (\text{clo}, x_1, \ldots, x_n)) \]

\[\text{as } \exists \alpha((\alpha \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket) \times \alpha) \]

\[\text{in clo} \]

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

\[ \llbracket \lambda x. M \rrbracket = \text{let code} = \lambda (\text{clo}, x). \]

\[ \llbracket M \rrbracket \]

\[ \text{let } (\_, x_1, \ldots, x_n) = \text{clo} \text{ in } \]

\[ \text{in } (\text{code}, x_1, \ldots, x_n) \]

\[ \llbracket M_1 M_2 \rrbracket = \text{let clo} = \llbracket M_1 \rrbracket \text{ in } \]

\[ \text{let code} = \text{proj}_0 \text{ clo} \text{ in } \]

\[ \text{code } (\text{clo}, \llbracket M_2 \rrbracket) \]

There are two difficulties:

- a closure is a tuple, whose \textit{first} field should be \textit{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 \textit{the closure itself} as its first argument.
Closure-passing closure conversion

There are two difficulties:

- a closure is a tuple, whose \textit{first} field should be \textit{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 \textit{the closure itself} as its first argument.

What type-theoretic mechanisms could we use to describe this?

- existential quantification over the \textit{tail} of a tuple (a.k.a. a \textit{row});
- \textit{recursive types}.
Tuples, rows, row variables

The standard tuple types that we have used so far are:

\[
\tau ::= \ldots | \Pi R \quad - \text{types} \\
R ::= \epsilon | (\tau; R) \quad - \text{rows}
\]

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:

\[
\tau ::= \ldots | \Pi R | \forall \rho. \tau | \exists \rho. \tau \quad - \text{types} \\
R ::= \rho | \epsilon | (\tau; R) \quad - \text{rows}
\]

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:

\[
\begin{align*}
\text{TUPLE} & \quad \forall i. \in [1, n] \quad \Gamma \vdash M_i : \tau_i \\
\Gamma & \vdash (M_1, \ldots, M_n) : \Pi (\tau_1; \ldots; \tau_n; \epsilon)
\end{align*}
\]

\[
\begin{align*}
\text{PROJ} & \quad \Gamma \vdash M : \Pi (\tau_1; \ldots; \tau_i; R) \\
\Gamma & \vdash \text{proj}_i M : \tau_i
\end{align*}
\]

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 \textit{parametric polymorphism}:

\[
\text{proj}_i : \forall \alpha._{1 \ldots i} \rho. \Pi (\alpha_1; \ldots; \alpha_i; \rho) \to \alpha_i
\]
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:

\[
\begin{array}{c}
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket \\
= \exists \rho. \\
\mu \alpha. \\
\Pi ( \\
(\alpha \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket; \\
\rho \\
)
\end{array}
\]

\(\rho\) describes the environment
\(\alpha\) is the concrete type of the closure
a tuple...
...that begins with a code pointer...
...and continues with the environment

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

**Question:** Why is it \(\exists \rho. \mu \alpha. \tau\) and not \(\mu \alpha. \exists \rho. \tau\)

*The type of the environment is fixed once for all and does not change at each recursive call.*

**Question:** Notice that \(\rho\) appears only once. Any comments?
Closure-passing closure conversion

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

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

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

$$\llbracket \lambda x:\llbracket\tau_1\rrbracket . M \rrbracket = \text{let code : } (\text{Clo}(\llbracket\Gamma\rrbracket) \times \llbracket\tau_1\rrbracket) \to \llbracket\tau_2\rrbracket = \lambda (\text{clo : Clo}(\llbracket\Gamma\rrbracket), x : \llbracket\tau_1\rrbracket).$$

$$\text{let } (\_, x_1, \ldots, x_n) : \text{UClo}[\Gamma] = \text{unfold clo in } \llbracket M \rrbracket \text{ in } \text{pack}[\Gamma], \text{fold(code, x}_1, \ldots, x_n) \text{ as } \exists \rho.\text{Clo}(\rho)$$

$$\llbracket M_1 M_2 \rrbracket = \text{let } \rho, \text{clo} = \text{unpack } \llbracket M_1 \rrbracket \text{ in } \text{let code : } (\text{Clo}(\rho) \times \llbracket\tau_1\rrbracket) \to \llbracket\tau_2\rrbracket = \text{proj}_0(\text{unfold clo}) \text{ in } \text{code(clo, } \llbracket M_2 \rrbracket )$$
In the closure-passing variant, recursive functions can be translated as:

\[
\left[ \mu f. \lambda x. M \right] = \text{let } code = \lambda (\text{clo}, x). \\
\text{let } f = \text{clo in} \\
\text{let } (\_, x_1, \ldots, x_n) = \text{clo in} \\
\left[ M \right] \\
\text{in } (\text{code}, x_1, \ldots, x_n)
\]

where \( \{x_1, \ldots, x_n\} = \text{fv}(\mu f. \lambda x. M) \).

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

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 invocation, 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

Let $\tau$ be $\forall \vec{\alpha}. \tau_1 \rightarrow \tau_2$ and $\Gamma_f$ be $f : \tau, \Gamma$ where $\vec{\beta} \not\in \Gamma$

$$\llbracket \mu f : \tau. \lambda x. M \rrbracket = \text{let code } = $$

$$\Lambda \vec{\beta}. \lambda (\text{clo} : Clo[\Gamma_f], x : [\tau_1]).$$

$$\text{let } (\_\text{code}, f, x_1, \ldots, x_n) : \forall \vec{\beta}. UClo([\Gamma_f]) =$$

$$\text{unfold clo in }$$

$$\llbracket M \rrbracket \text{ in }$$

$$\text{let rec clo : } \forall \vec{\beta}. \exists \rho. Clo(\rho) = \Lambda \vec{\beta}.$$

$$\text{pack } [\Gamma], (\text{fold } (\text{code } \vec{\beta}, \text{clo}, x_1, \ldots, x_n)) \text{ as } \exists \rho. Clo(\rho)$$

$$\text{in clo}$$

Remind that $Clo(R)$ abbreviates $\mu \alpha. \Pi ((\alpha \times [\tau_1]) \rightarrow [\tau_2]; R)$. Hence, $\vec{\beta}$ are free variables of $Clo(R)$.

Here, a polymorphic recursive function is \textit{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

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:

\[
\llbracket M \rrbracket = \begin{array}{l}
\text{let } code_i = \lambda (env, x). \\
\text{let } (f_1, f_2, x_1, \ldots, x_n) = env \text{ in} \\
\llbracket M_i \rrbracket \\
\text{in} \\
\text{let } \text{rec } clo_1 = (code_1, (clo_1, clo_2, x_1, \ldots, x_n)) \\
\text{and } clo_2 = (code_2, (clo_1, clo_2, x_1, \ldots, x_n)) \text{ in} \\
clo_1, clo_2
\end{array}
\]
Mutually recursive functions

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:

\[
\begin{align*}
[M] &= \begin{array}{l}
\text{let } code_i = \lambda (env, x).
\text{let } (f_1, f_2, x_1, \ldots, x_n) = env \text{ in }
[M_i]
\end{array} \\
\text{in }
\begin{array}{l}
\text{let } rec \text{ env } = (clo_1, clo_2, x_1, \ldots, x_n) \\
\text{and } clo_1 = (code_1, env) \\
\text{and } clo_2 = (code_2, env) \text{ in }
\end{array} \\
clo_1, clo_2
\end{align*}
\]
Mutually recursive functions

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:

\[
\text{let } code_i = \lambda (clo, x). \quad \text{let } (_, f_1, f_2, x_1, \ldots, x_n) = \text{clo in } [M_i] \\
\text{in } \quad \text{let rec } clo_1 = (code_1, clo_1, clo_2, x_1, \ldots, x_n) \quad \text{and } clo_2 = (code_2, clo_1, clo_2, x_1, \ldots, x_n) \\\n\text{in } clo_1, clo_2
\]

**Question:** Can we share the closures \( c_1 \) and \( c_2 \) in case \( n \) is large?
Mutually recursive functions

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:

\[
\begin{align*}
&\text{let } code_1 = \lambda(clo, x).
&\text{let } (\_code_1, \_code_2, f_1, f_2, x_1, \ldots, x_n) = clo \ \text{in } [M_1] \ \text{in} \\
&\text{let } code_2 = \lambda(clo, x).
&\text{let } (\_code_2, f_1, f_2, x_1, \ldots, x_n) = clo \ \text{in } [M_2] \ \text{in} \\
&\text{let rec } clo_1 = (code_1, code_2, clo_1, clo_2, x_1, \ldots, x_n) \ \text{and } clo_2 = clo_1.tail \\
&\text{in } clo_1, clo_2
\end{align*}
\]

- \( clo_1.tail \) returns a pointer to the tail \((code_2, clo_1, clo_2, x_1, \ldots, x_n)\) of \( clo_1 \) without allocating a new tuple.
- This is only possible with some support from the GC (and extra-complexity and runtime cost for GC)
Optimizing representations

Can closure passing and environment passing be mixed?

No because the calling-convention \( (i.e., \text{the encoding of application}) \) must be uniform.

However, there is some flexibility in the representation of the closure. For instance, the following change is completely local:

\[
\left[ \lambda x. M \right] = \text{let } \text{code} = \lambda (\text{clo}, x). \quad \text{let } (-, \left( x_1, \ldots, x_n \right)) = \text{clo} \text{ in } \left[ M \right] \text{ in } \left( \text{code}, \left( x_1, \ldots, x_n \right) \right)
\]

\[
\left[ M_1 \; M_2 \right] = \text{let } \text{clo} = \left[ M_1 \right] \text{ in } \text{let } \text{code} = \text{proj}_0 \; \text{clo} \text{ in } \text{code} \left( \text{clo}, \left[ M_2 \right] \right)
\]

Applications? When many definitions share the same closure, the closure (or part of it) may be shared.
Encoding of objects

The closure-passing representation of mutually recursive functions is similar to the representations of objects in the object-as-record-of-functions paradigm:

A class definition is an object generator:

```java
class c (x_1, \ldots x_q) {
    \text{meth} \; m_1 = M_1 \\
    \ldots \\
    \text{meth} \; m_p = M_p 
}
```

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:

\[
\text{let } m = \\
\quad \text{let } m_1 = \lambda (m, x_1, \ldots, x_q). M_1 \text{ in} \\
\quad \quad \quad \quad \ldots \\
\quad \text{let } m_p = \lambda (m, x_1, \ldots, x_q). M_p \text{ in} \\
\quad \quad \{ m_1, \ldots, m_p \} \text{ in} \\
\quad \lambda x_1 \ldots x_q. (m, x_1, \ldots x_q)
\]

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

\[
(proj_0 p).m_i p
\]

How can we type the encoding?
Typed encoding of objects

Let $\tau_i$ be the type of $M_i$, and row $R$ describe the types of $(x_1, \ldots x_q)$.

Let $Clo(R)$ be $\mu\alpha.\Pi(\{(m_i : \alpha \to \tau_i)_{i\in I}\}; R)$ and $UClo(R)$ its unfolding.

Fields $R$ are hidden in an existential type $\exists \rho. \mu\alpha.\Pi(\{(m_i : \alpha \to \tau_i)_{i\in I}\}; \rho)$:

\[
\begin{align*}
let m &= \{ \\
    m_1 &= \lambda(m, x_1, \ldots x_q : UClo(R)).[M_1] \\
    \vdots \\
    m_p &= \lambda(m, x_1, \ldots x_q : UClo(R)).[M_p] \\
\} \ in \\
\lambda x_1. \ldots \lambda x_q. \text{pack } R, \text{fold } (m, x_1, \ldots x_q) \ as \ \exists \rho. \ (M, \rho)
\end{align*}
\]

Calling a method of an object $p$ of type $M$ is

\[
p\#m_i \overset{\triangle}{=} let \ \rho, z = \text{unpack } p \ in \ (\text{proj}_0 \ \text{unfold } z).m_i \ z
\]

An object has a recursive type but it is \textit{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.

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 na"ıve 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

- Presentation

- Expressiveness

- Beyond $F^\omega$
Polymorphism in System F

**Simply-typed \( \lambda \)-calculus**

- no polymorphism
- many functions must be duplicated at different types

**Via ML style (let-binding) polymorphism**

- Considerable improvement by avoiding most of code duplication.
- ML has also local let-polymorphism (less critical).
- Still, ML is lacking existential types—compensated by modules and sometimes lacking higher-rank polymorphism

**System F brings much more expressiveness**

- Existential types—allows for type abstraction
- First-class universal types
- Allows for encoding of data structures and more programming patterns

Still, limited...
Limits of System F

Map on pairs, say pair\_map, has the following incompatible types:

\[ \forall \alpha_1. \forall \alpha_2. (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2 \]
\[ \forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2 \]

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

It is missing the ability to describe the types of functions

- that are polymorphic in one parameter
- but whose domain and codomain are otherwise arbitrary

i.e. of the form \( \forall \alpha. \tau[\alpha] \to \sigma[\alpha] \) for arbitrary one-hole types \( \tau \) and \( \sigma \).

We just need to abstract over such contexts, i.e., over type functions:

\[ \forall \varphi. \forall \psi. \forall \alpha_1. \forall \alpha_2. (\forall \alpha. \varphi \alpha \to \psi \alpha) \to \varphi \alpha_1 \to \varphi \alpha_2 \to \psi \alpha_1 \times \psi \alpha_2 \]
From System F to System F\(^{\omega}\)

We introduce kinds \(\kappa\) for types (with a single kind \(*\) to stay in System F)

Well-formedness of types becomes \(\Gamma \vdash \tau : *\):

\[
\begin{align*}
\vdash \Gamma \alpha : \kappa \in \Gamma \\
\Gamma \vdash \alpha : \kappa \\
\Gamma \vdash \tau_1 : * \\
\Gamma \vdash \tau_2 : * \\
\Gamma \vdash \tau_1 \rightarrow \tau_2 : * \\
\Gamma \vdash \forall \alpha : \kappa. \tau : * \\
\end{align*}
\]

We add and check kinds on type abstractions and type applications:

\[
\begin{align*}
\vdash \emptyset \\
\vdash \Gamma \alpha \notin \text{dom}(\Gamma) \\
\vdash \Gamma, \alpha : \kappa \\
\vdash \Gamma \tau \notin \text{dom}(\Gamma) \\
\vdash \Gamma, x : \tau \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, \alpha : \kappa \vdash M : \tau \\
\Gamma \vdash \Lambda \alpha : \kappa . M : \forall \alpha : \kappa . \tau \\
\Gamma \vdash M \tau' : \forall \alpha : \kappa . \tau \\
\end{align*}
\]

So far, this is an equivalent formalization of System F
From System F to System $F^\omega$

Redefine kinds as

$$\kappa ::= \ast \mid \kappa \Rightarrow \kappa$$

New types

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

Typing of expressions is up to type equivalence:

$$\Gamma \vdash M : \tau \quad \tau \equiv_\beta \tau'$$

Remark

$$\Gamma \vdash M : \tau \implies \Gamma \vdash \tau : \ast$$
$F^\omega$, static semantics

(altogether on one slide)

### Syntax

\[\begin{align*}
\kappa & ::= * \mid \kappa \Rightarrow \kappa \\
\tau & ::= \alpha \mid \tau \to \tau \mid \forall \alpha. \tau \mid \lambda \alpha. \tau \mid \tau \tau \\
M & ::= x \mid \lambda x: \tau. M \mid M M \mid \Lambda \alpha. M \mid M \tau
\end{align*}\]

### Kinding rules

\[
\begin{align*}
\vdash \Gamma \\
\vdash \emptyset \\
\vdash \Gamma, \alpha : \kappa \\
\vdash \Gamma, x : \tau \\
\vdash \forall \alpha. \tau : \kappa \\
\vdash \lambda \alpha. \tau : \kappa_1 \Rightarrow \kappa_2 \\
\vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \\
\vdash \tau_2 : \kappa_2 \\
\vdash \tau_1 \Rightarrow \tau_2 : \kappa_1
\end{align*}
\]

### Typing rules

\[
\begin{align*}
\text{VAR} & \\
x : \tau \in \Gamma \\
\Gamma \vdash x : \tau \\
\text{ABS} & \\
\Gamma, x : \tau_1 \vdash M : \tau_2 \\
\Gamma \vdash \lambda x : \tau_1 . M : \tau_1 \to \tau_2 \\
\text{TABS} & \\
\Gamma, \alpha : \kappa \vdash M : \tau \\
\Gamma \vdash \Lambda \alpha. M : \forall \alpha. \tau \\
\text{TAPP} & \\
\Gamma \vdash M : \forall \alpha. \tau \\
\Gamma \vdash \tau' : \kappa \\
\Gamma \vdash M \tau' : [\alpha \mapsto \tau'] \tau \\
\text{TEquiv} & \\
\Gamma \vdash M : \tau \\
\Gamma \vdash \tau \equiv_\beta \tau' \\
\Gamma \vdash M : \tau'
\end{align*}
\]

With implicit kinds

- \[\kappa \mapsto \alpha / \text{dividend} \langle \text{0} \rangle\]
- \[\tau \mapsto \alpha / \text{dividend} \langle \text{0} \rangle \rightarrow \tau \mapsto \forall \alpha. \tau \mapsto \lambda \alpha. \tau \mapsto \tau \tau\]
- \[\alpha \mapsto \kappa \in \Gamma \]
- \[\Gamma \vdash \alpha : \kappa \]
- \[\Gamma \vdash \tau_1 : \kappa \Rightarrow \kappa_2 \]
- \[\Gamma \vdash \tau_2 : \kappa_2 \Rightarrow \kappa_1 \]
- \[\Gamma \vdash \tau_1 \Rightarrow \tau_2 : \kappa_1 \]
\( F^\omega \), dynamic semantics

The semantics is unchanged (modulo kind annotations in terms)

\[
V ::= \lambda x:\tau. M \mid \Lambda \alpha :: \kappa. V
\]

\[
E ::= [] M \mid V [] \mid [] \tau \mid \Lambda \alpha :: \kappa. []
\]

\[
(\lambda x:\tau. M) V \rightarrow [x \mapsto V]M
\]

\[
(\Lambda \alpha :: \kappa. V) \tau \rightarrow [\alpha \mapsto \tau]V
\]

No type reduction

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

Kinds are erasable

- Kinds are preserved by type and term reduction.
- Kinds may be ignored during reduction—or erased prior to reduction.
Properties

Main properties are preserved. Proofs are similar to those for System F.

Type soundness

- Subject reduction
- Progress

Termination of reduction

(In the absence of construct for recursion.)

Typechecking is decidable

- This requires reduction at the level of types to check type equality
- Can be done by putting types in normal forms using full reduction (on types only), or just head normal forms.
Type reduction

Used for typechecking to check type equivalence $\equiv$

Full reduction of the simply typed $\lambda$-calculus

$$(\lambda\alpha. \tau) \sigma \rightarrow [\alpha \mapsto \tau]\sigma$$

applicable in any type context.

Type reduction preserve types: this is subject reduction for simply-typed $\lambda$-calculus (when terms are now used as types), but for full reduction

(we have only proved it for CBV).

It is a key that reduction terminates.

(which again, we have only proved for CBV.)
Contents

- Presentation
- Expressiveness
- Beyond $F^\omega$
Expressiveness

More polymorphism
- pair_map

Abstraction over type operators
- monads
- encoding of existentials

Other encodings
- non regular datatypes
- equality
- modules
Pair map in $F^\omega$ (with implicit kinds) \[ \lambda fxy. (f x, f y) \]

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

\[
\Lambda \varphi. \Lambda \psi. \Lambda \alpha_1. \Lambda \alpha_2. \\
\lambda (f : \forall \alpha. \varphi \alpha \rightarrow \psi \alpha). \lambda x : \varphi \alpha_1. \lambda y : \varphi \alpha_2. (f \alpha_1 x, f \alpha_2 y)
\]
call it pair_map of type:

\[
\forall \varphi. \forall \psi. \forall \alpha_1. \forall \alpha_2. \\
(\forall \alpha. \varphi \alpha \rightarrow \psi \alpha) \rightarrow \varphi \alpha_1 \rightarrow \varphi \alpha_2 \rightarrow \psi \alpha_1 \times \psi \alpha_2
\]

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

\[
\Lambda \alpha_1. \Lambda \alpha_2. \lambda f : \alpha_1 \rightarrow \alpha_2. \text{pair}_\text{map} (\lambda \alpha. \alpha_1) (\lambda \alpha. \alpha_2) \alpha_1 \alpha_2 (\Lambda \gamma. f) \] 

\[
: \forall \alpha_1. \forall \alpha_2. (\forall \gamma. \alpha_1 \rightarrow \alpha_2) \rightarrow \alpha_1 \rightarrow \alpha_1 \rightarrow \alpha_2 \times \alpha_2
\]

\text{pair}_\text{map} (\lambda \alpha. \alpha) (\lambda \alpha. \alpha) 

\[
: \forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_1 \times \alpha_2
\]

Still, the type of \text{pair}_\text{map} is not principal: $\varphi$ and $\psi$ could depend on two variables, i.e. be of kind $\star \Rightarrow \star \Rightarrow \star$, or many other kinds...
Pair map in $F^\omega$ (with implicit kinds) $\lambda f x y. (f x, f y)$

Abstract over (one parameter) type \textit{functions} (e.g. of kind $\star \rightarrow \star$)

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

call it \texttt{pair_map} of type:

$$\forall \varphi. \forall \psi. \forall \alpha_1. \forall \alpha_2. \left((\forall \alpha. \varphi \alpha \rightarrow \psi \alpha) \rightarrow \varphi \alpha_1 \rightarrow \varphi \alpha_2 \rightarrow \psi \alpha_1 \times \psi \alpha_2\right)$$

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

$$\Lambda \alpha_1. \Lambda \alpha_2. \lambda f : \alpha_1 \rightarrow \alpha_2. \texttt{pair_map} \ (\lambda \alpha. \alpha_1) \ (\lambda \alpha. \alpha_2) \ \alpha_1 \ \alpha_2 \ (\Lambda \gamma. f)$$

$$: \forall \alpha_1. \forall \alpha_2. \left(\forall \gamma. \alpha_1 \rightarrow \alpha_2\right) \rightarrow \alpha_1 \rightarrow \alpha_1 \rightarrow \alpha_2 \times \alpha_2$$

\texttt{pair_map} $(\lambda \alpha. \alpha) \ (\lambda \alpha. \alpha)$

$$: \forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_1 \times \alpha_2$$

Still, the type of \texttt{pair_map} is not principal: $\varphi$ and $\psi$ could depend on two variables, \textit{i.e.} be of kind $\star \Rightarrow \star \Rightarrow \star$, or many other kinds...
Abstracting over type operators

**Type of monads** Given a type operator $\varphi$, a monad is given by a pair of two functions of the following type (satisfying certain laws).

$$M \triangleq \lambda \varphi. \quad \{ \text{ret} : \forall \alpha. \alpha \to \varphi\alpha; \quad \text{bind} : \forall \alpha. \forall \beta. \varphi\alpha \to (\alpha \to \varphi\beta) \to \varphi\beta \}$$

: $(\ast \Rightarrow \ast) \Rightarrow \ast$

(Notice that $M$ is itself of higher kind)

**A generic map function:** can then be defined:

$$fmap \triangleq \lambda m. \lambda f. \lambda x. \quad m.\text{bind} \ x \ (\lambda x. m.\text{ret} \ (f \ x))$$

: $\forall \varphi. M \varphi \to \forall \alpha. \forall \beta. (\alpha \to \beta) \to \varphi\alpha \to \varphi\beta$
Abstracting over type operators

**Type of monads** Given a type operator \( \varphi \), a monad is given by a pair of two functions of the following type (satisfying certain laws).

\[
\mathcal{M} \triangleq \lambda \varphi. \\
\{ \text{ret} : \forall \alpha. \alpha \to \varphi \alpha; \text{bind} : \forall \alpha. \forall \beta. \varphi \alpha \to (\alpha \to \varphi \beta) \to \varphi \beta \} \]

: \( \ast \Rightarrow \ast \Rightarrow \ast \)

(Notice that \( \mathcal{M} \) is itself of higher kind)

**A generic map function:** can then be defined:

\[
fmap \triangleq \lambda m. \\
\lambda f. \lambda x. m.\text{bind} x (\lambda x. m.\text{ret} (f x)) \]

: \( \forall \varphi. \mathcal{M} \varphi \to \forall \alpha. \forall \beta. (\alpha \to \beta) \to \varphi \alpha \to \varphi \beta \)
Abstracting over type operators

Available in Haskell

- $\varphi\alpha$ is treated as a type app($\varphi$, $\alpha$) where app : $(\kappa_1 \Rightarrow \kappa_2) \Rightarrow \kappa_1 \Rightarrow \kappa_2$
- No $\beta$-reduction at the level of types: $\varphi\alpha = \psi\beta \iff \varphi = \psi \land \alpha = \beta$
- Compatible with type inference (first-order unification)
- Since there is no type $\beta$-reduction, this is not $F^\omega$.

Encodable in OCaml with modules

- See [Yallop and White, 2014] (and also [Kiselyov])
- As in Haskell, the encoding does not handle type $\beta$-reduction
- As a counterpart, this allows for type inference at higher kinds (as in Haskell).
Encoding of existentials

We saw

\[ [\exists \alpha. \tau] = \forall \beta. (\forall \alpha. \tau \to \beta) \to \beta \]

Hence,

\[ \left[ \text{pack}_{\exists \alpha. \tau} \right] \triangleq \Lambda \alpha. \lambda x : [\tau]. \Lambda \beta. \lambda k : \forall \alpha. ([\tau] \to \beta). k \alpha x \]

This requires a different code for each type \( \tau \)

To have a unique code, we just abstract over \( \lambda \alpha. \tau \), i.e. \( \varphi \):

In System \( F^\omega \), we may defined

\[ \left[ \text{pack}_{\kappa} \right] = \Lambda \varphi. \Lambda \alpha. \lambda x : \varphi \alpha. \Lambda \beta. \lambda k : \forall \alpha. (\varphi \alpha \to \beta). k \alpha x \]

(omitting kinds)

Allows existentials at higher kinds!
Encoding of existentials

We saw

\[ [\exists \alpha. \tau] = \forall \beta. (\forall \alpha. \tau \to \beta) \to \beta \]

Hence,

\[ [\text{pack}_{\exists \alpha. \tau}] \triangleq \Lambda \alpha. \lambda x : [\tau]. \Lambda \beta. \lambda k : \forall \alpha. ([\tau] \to \beta). k \alpha x \]

This requires a different code for each type \( \tau \)

To have a unique code, we just abstract over \( \lambda \alpha. \tau \), i.e. \( \varphi \):

In System \( F^\omega \), we may defined

\[ [\text{pack}_{\kappa}] = \Lambda \varphi. \Lambda \alpha. \lambda x : \varphi \alpha. \Lambda \beta. \lambda k : \forall \alpha. (\varphi \alpha \to \beta). k \alpha x \]

(omitting kinds)

Allows existentials at higher kinds!
Exploiting kinds

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

$$\tau = \alpha \mid \lambda \alpha : \kappa . \tau \mid \tau \tau \mid \tau \to \tau \mid g$$

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

- $\times : \star \Rightarrow \star \Rightarrow \star$
- $+ : \star \Rightarrow \star \Rightarrow \kappa$
- $\forall \kappa :: (\kappa \Rightarrow \star) \Rightarrow \star$
- $\exists \kappa :: (\kappa \Rightarrow \star) \Rightarrow \star$
- $(\tau \times \tau) \triangleq (\times) \ τ_1 \ τ_2$
- $(\tau + \tau) \triangleq (+) \ τ_1 \ τ_2$
- $\forall \varphi : \kappa . \tau \triangleq \forall \kappa (\lambda \varphi : \kappa \Rightarrow \star . \tau)$
- $\exists \varphi : \kappa . \tau \triangleq \exists \kappa (\lambda \varphi : \kappa \Rightarrow \star . \tau)$

In fact $F^\omega$ could be extended with kind abstraction:

- $\forall :: \forall \kappa . (\kappa \Rightarrow \star) \Rightarrow \star$
- $\exists :: \forall \kappa . (\kappa \Rightarrow \star) \Rightarrow \star$
- $\forall \varphi : \kappa . \tau \triangleq \forall \kappa (\lambda \varphi : \kappa \Rightarrow \star . \tau)$
- $\exists \varphi : \kappa . \tau \triangleq \exists \kappa (\lambda \varphi : \kappa \Rightarrow \star . \tau)$

When kinds are inferred:

- $\forall \varphi . \tau \triangleq \forall (\lambda \varphi . \tau)$
- $\exists \varphi . \tau \triangleq \exists (\lambda \varphi . \tau)$
Church encoding of regular ADT

\[
\text{type} \quad \text{List} \, \alpha = \\
\mid \text{Nil} : \forall \alpha. \text{List} \, \alpha \\
\mid \text{Cons} : \forall \alpha. \alpha \rightarrow \text{List} \, \alpha \rightarrow \text{List} \, \alpha
\]

Church encoding (CPS style) in System F

\[
\text{List} \overset{\triangle}{=} \lambda \alpha. \forall \beta. \beta \rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta
\]

\[
\text{Nil} \overset{\triangle}{=} \lambda n. \lambda c. n : \forall \alpha. \text{List} \, \alpha
\]

\[
\text{Cons} \overset{\triangle}{=} \lambda x. \lambda \ell. \lambda n. \lambda c. x (\ell \beta n c) : \forall \alpha. \alpha \rightarrow \text{List} \, \alpha \rightarrow \text{List} \, \alpha
\]

\[
\text{fold} \overset{\triangle}{=} \lambda n. \lambda c. \lambda \ell. \ell \beta n c
\]

Actually not enhanced!

Be aware of useless over-generalization!

For regular ADTs, all uses of \( \varphi \) are \( \varphi \alpha \).
Hence, \( \forall \alpha. \forall \varphi. \tau [\varphi \alpha] \) is not more general than \( \forall \alpha. \forall \beta. \tau [\beta] \)
Church encoding of regular ADT

\[
\text{type } \ List \ \alpha = \\
\mid \ Nil \ : \ \forall \alpha. \ List \ \alpha \\
\mid \ Cons : \ \forall \alpha. \ \alpha \to \ List \ \alpha \to \ List \ \alpha
\]

Church encoding (CPS style) in System F

\[
List \ \triangleq \ \lambda \alpha. \ \forall \beta. \ \beta \to (\alpha \to \beta \to \beta) \to \beta \\
Nil \ \triangleq \ \lambda n. \ \lambda c. \ n \ : \ \forall \alpha. \ List \ \alpha \\
Cons \ \triangleq \ \lambda x. \ \lambda \ell. \ \lambda n. \ \lambda c. \ c \ x (\ell \beta n c) \ : \ \forall \alpha. \ \alpha \to \ List \ \alpha \to \ List \ \alpha
\]

\[
fold \ \triangleq \ \lambda n. \ \lambda c. \ \lambda \ell. \ \ell \beta n c
\]

Actually not enhanced! Be aware of useless over-generalization!

For regular ADTs, all uses of \( \varphi \) 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

\[
\text{type } \text{Seq } \alpha =
\begin{align*}
| \text{Nil} & : \forall \alpha. \text{Seq } \alpha \\
| \text{Zero} & : \forall \alpha. \text{Seq } (\alpha \times \alpha) \rightarrow \text{Seq } \alpha \\
| \text{One} & : \forall \alpha. \alpha \rightarrow \text{Seq } (\alpha \times \alpha) \rightarrow \text{Seq } \alpha
\end{align*}
\]

Encoded as:

\[
\text{Seq } \triangleq \lambda \alpha. \forall \varphi. (\forall \alpha. \varphi \alpha) \rightarrow (\forall \alpha. \varphi (\alpha \times \alpha) \rightarrow \varphi \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \varphi (\alpha \times \alpha) \rightarrow \varphi \alpha) \rightarrow \varphi \alpha
\]

\[
\text{Nil } \triangleq \lambda n. \lambda z. \lambda s. n : \forall \alpha. \text{Seq } \alpha
\]

\[
\text{Zero } \triangleq \lambda l. \lambda n. \lambda z. \lambda s. z (l n z s) : \forall \alpha. \text{Seq } (\alpha \times \alpha) \rightarrow \text{Seq } \alpha
\]

\[
\text{One } \triangleq \lambda x. \lambda l. \lambda n. \lambda z. \lambda s. s x (l n z s) : \forall \alpha. \alpha \rightarrow \text{Seq } (\alpha \times \alpha) \rightarrow \text{Seq } \alpha
\]

\[
\text{fold } \triangleq \lambda n. \lambda z. \lambda s. \lambda l. l n z s
\]

Cannot be simplified! Indeed \(\varphi\) is applied to both \(\alpha\) and \(\alpha \times \alpha\).

Non regular ADTs cannot be encoded in System F.
Church encoding of *non*-regular ADTs

**Okasaki’s Seq**

\[
\text{type } \text{Seq} \, \alpha =
\begin{align*}
| \text{Nil} & : \forall \alpha. \text{Seq} \, \alpha \\
| \text{Zero} & : \forall \alpha. \text{Seq} \, (\alpha \times \alpha) \rightarrow \text{Seq} \, \alpha \\
| \text{One} & : \forall \alpha. \alpha \rightarrow \text{Seq} \, (\alpha \times \alpha) \rightarrow \text{Seq} \, \alpha
\end{align*}
\]

Encoded as:

\[
\text{Seq} \triangleq \lambda \alpha. \forall \varphi. (\forall \alpha. \varphi \, \alpha) \rightarrow (\forall \alpha. \varphi (\alpha \times \alpha) \rightarrow \varphi \, \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \varphi (\alpha \times \alpha) \rightarrow \varphi \, \alpha) \rightarrow \varphi \, \alpha
\]

\[
\text{Nil} \triangleq \lambda n. \lambda z. \lambda s. n : \forall \alpha. \text{Seq} \, \alpha
\]

\[
\text{Zero} \triangleq \lambda \ell. \lambda n. \lambda z. \lambda s. z (\ell \, n \, z \, s) : \forall \alpha. \text{Seq} \, (\alpha \times \alpha) \rightarrow \text{Seq} \, \alpha
\]

\[
\text{One} \triangleq \lambda x. \lambda \ell. \lambda n. \lambda z. \lambda s. s \, x (\ell \, n \, z \, s) : \forall \alpha. \alpha \rightarrow \text{Seq} \, (\alpha \times \alpha) \rightarrow \text{Seq} \, \alpha
\]

\[
f \text{fold} \triangleq \lambda n. \lambda z. \lambda s. \lambda \ell. \ell \, n \, z \, s
\]

Cannot be simplified! Indeed \(\varphi\) is applied to both \(\alpha\) and \(\alpha \times \alpha\).

Non regular ADTs cannot be encoded in System F.
Equality

\[\text{module } \text{Eq : EQ = struct} \]
\[\text{type } (\alpha, \beta) \text{ eq = Eq : } (\alpha, \alpha) \text{ eq} \]
\[\text{let } \text{coerce } (\text{type } a) (\text{type } b) (\text{ab : (a,b) eq}) (x : a) : b = \text{let } \text{Eq = ab in } x \]
\[\text{let } \text{refl : (} \alpha, \alpha \text{) eq = Eq} \]

(* all these are propagation and automatic with GADTs *)
\[\text{let } \text{symm } (\text{type } a) (\text{type } b) (\text{ab : (a,b) eq}) : (b,a) \text{ eq = let } \text{Eq = ab in ab} \]
\[\text{let } \text{trans } (\text{type } a) (\text{type } b) (\text{type } c) \]
\[\text{ (ab : (a,b) eq) (bc : (b,c) eq) : (a,c) \text{ eq = let } \text{Eq = ab in bc} \]
\[\text{let } \text{lift } (\text{type } a) (\text{type } b) (\text{ab : (a,b) eq}) : (a \text{ list, b list}) \text{ eq = \text{let } Eq = ab in Eq} \]

end
Equality

Leibnitz equality in $F^\omega$

$$Eq\;\alpha\;\beta \equiv \forall \varphi.\;\varphi\alpha \rightarrow \varphi\beta$$

$$\begin{align*}
Eq & \triangleq \lambda\alpha.\;\lambda\beta.\;\forall \varphi.\;\varphi\alpha \rightarrow \varphi\beta \\
coerce & \triangleq \lambda p.\;\lambda x.\; p\;x \\
\quad : \forall \alpha.\;\forall \beta.\;Eq\;\alpha\;\beta \rightarrow \alpha \rightarrow \beta \\
refl & \triangleq \lambda x.\; x \\
\quad : \forall \alpha.\;\forall \varphi.\;\varphi\alpha \rightarrow \varphi\alpha \equiv \forall \alpha.\;Eq\;\alpha\;\alpha \\
\symm & \triangleq \lambda p.\; p\;(refl) \\
\quad : \forall \alpha.\;\forall \beta.\;Eq\;\alpha\;\beta \rightarrow Eq\;\beta\;\alpha \\
\trans & \triangleq \lambda p.\;\lambda q.\; q\;p \\
\quad : \forall \alpha.\;\forall \beta.\;\forall \gamma.\;Eq\;\alpha\;\beta \rightarrow Eq\;\beta\;\gamma \rightarrow Eq\;\alpha\;\gamma Equiv Eq\;\alpha\;\beta \rightarrow Eq\;\alpha\;\gamma \\
\lift & \triangleq \lambda p.\; p\;(refl) \\
\quad : \forall \alpha.\;\forall \beta.\;\forall \varphi.\;Eq\;\alpha\;\beta \rightarrow Eq\;\varphi\alpha\;\varphi\beta : Eq\,(\varphi\alpha)\,(\varphi\alpha) \rightarrow Eq\,(\varphi\alpha)\,(\varphi\beta) \end{align*}$$
Equality

Leibnitz equality in $F^\omega$

$$Eq \alpha \beta \equiv \forall \varphi. \varphi \alpha \rightarrow \varphi \beta$$

\[ Eq \equiv \lambda \alpha. \lambda \beta. \forall \varphi. \varphi \alpha \rightarrow \varphi \beta \]

\[ coerce \equiv \lambda p. \lambda x. p \ x \]
\[ \text{ : } \forall \alpha. \forall \beta. Eq \alpha \beta \rightarrow \alpha \rightarrow \beta \]

\[ refl \equiv \lambda x. \ x \]
\[ \text{ : } \forall \alpha. \forall \varphi. \varphi \alpha \rightarrow \varphi \alpha \equiv \forall \alpha. Eq \alpha \alpha \]

\[ symm \equiv \lambda p. p \ (\text{refl}) \]
\[ \text{ : } \forall \alpha. \forall \beta. Eq \alpha \beta \rightarrow Eq \beta \alpha \]

\[ trans \equiv \lambda p. \lambda q. q \ p \]
\[ \text{ : } \forall \alpha. \forall \beta. \forall \gamma. Eq \alpha \beta \rightarrow Eq \beta \gamma \rightarrow Eq \alpha \gamma \]

\[ lift \equiv \lambda p. p \ (\text{refl}) \]
\[ \text{ : } \forall \alpha. \forall \beta. \forall \varphi. Eq \alpha \beta \rightarrow Eq (\varphi \alpha) (\varphi \beta) \]

\[ : Eq (\varphi \alpha) (\varphi \alpha) \rightarrow Eq (\varphi \alpha) (\varphi \beta) \]
 Equality

Leibnitz equality in $F^\omega$

We implemented parts of the coercions of System Fc.

- We do not have decomposition of equalities (the inverse of Lift).
- This requires injectivity of type operators, which is not given.
- Equivalences and liftings must be written explicitly, while they are implicit with GADTs.

Some GATDs can be encoded, using equality plus existential types.
Contents

- Presentation
- Expressiveness
- Beyond $F^\omega$
A hierarchy of type systems

Kinds have a rank:

- the base kind $\ast$ is of rank 1
- kinds $\ast \Rightarrow \ast$ and $\ast \Rightarrow \ast \Rightarrow \ast$ have rank 2. They are the kinds of type functions taking type parameters of base kind.
- kind $\ast \Rightarrow \ast \Rightarrow \ast$ has rank 3—it is a type function whose parameter is itself a simple type function (of rank 1).
- more generally, $\text{rank} (\kappa_1 \Rightarrow \kappa_2) = \max(1 + \text{rank} \kappa_1, \text{rank} \kappa_2)$

This defines a sequence $F^1 \subseteq F^2 \subseteq F^3 \ldots \subseteq F^\omega$ of type systems of increasing expressiveness, where $F^n$ only uses kinds of rank $n$, whose limit is $F^\omega$ and where System F is $F^1$.

(Ranks are sometimes shifted by one, starting with $F = F^2$.)

Most examples in practice (and those we wrote) are in $F^2$, just above $F$. 
Extensions

Abstraction over kinds?

\[ \forall (\varphi :: * \Rightarrow *). \forall (\psi :: * \Rightarrow *). \forall (\alpha_1 :: *). \forall (\alpha_2 :: *). \\
    (\forall (\alpha :: *). \varphi \alpha \rightarrow \psi \alpha) \rightarrow \varphi \alpha_1 \rightarrow \varphi \alpha_2 \rightarrow \psi \alpha_1 \times \psi \alpha_2 \]

Motivation: `pair_map` does not have a principal type.
\( F^\omega \) with several base kinds

We could have several base kinds, e.g. \( * \) and \( field \) with type constructors:

\[
\begin{align*}
\text{filled} & : * \Rightarrow field \\
\text{empty} & : field
\end{align*}
\]

\[ box : field \Rightarrow * \]

Prevents ill-formed types such as \( box (\alpha \rightarrow filled \alpha) \).

This allows to build values \( v \) of type \( box \theta \) where \( \theta \) of kind \( field \) statically tells whether \( v \) is \( filled \) with a value of type \( \tau \) or \( empty \).

**Application:**

This is used in OCaml for rows of object types, but kinds are hidden to the user:

\[
\text{let get (x : \langle get : \alpha; .. \rangle)} : \alpha = x \# \text{get}
\]

The dots “..” here stand for a variable of another base kind (representing a row of types).
System $F^\omega$ 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 $\lambda$-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 $\ast$. 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 $F^\omega$ 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 $\lambda$-calculus with recursive types, equivalent to the untyped $\lambda$-calculus—every term is typable. Reduction of types does not terminate and type equality is ill-defined.

A solution proposed by Pottier [2011] is to force recursive kinds to be productive, reusing an idea from an [Nakano, 2000, 2001] for controlling recursion on terms, but pushing it one level up. Type equality becomes well-defined and semi-decidable.

The extension has been used to show that references in System F can be translated away in $F^\omega$ with guarded recursive kinds.
Generative functors can be encoded with existential types.

A functor $F$ has a type of the form:

$$\forall \bar{\alpha}. \tau[\bar{\alpha}] \rightarrow \exists \bar{\beta}. \sigma[\bar{\alpha}, \bar{\beta}]$$

Where:

- $\tau[\bar{\alpha}]$ represents the signature of the argument with some abstract types $\bar{\alpha}$.
- $\exists \bar{\beta}. \sigma[\bar{\alpha}, \bar{\beta}]$ represents the signature of the result of the functor application.
- That is, the abstract types $\bar{\alpha}$ are those taken from and shared with the argument.
- Conversely $\bar{\beta}$ are the abstract types created by the application, and have fresh identities independent of the argument.
- Two successive applications with the same argument (hence the same $\alpha$) will create two signatures with incompatible abstract types $\bar{\beta}_1$ and $\bar{\beta}_2$, once the existential is open.

Two applications of $F$ with the same argument:

Let module $\text{Z} = F(X)$ be understood as:

$$\text{let } \bar{\beta}, \text{Z} = \text{unpack}(F(X)) \text{ in } \ldots$$
Encoding ML modules with applicative functors

Applicative functors can be encoded with higher-order existential types.

A functor \( F \) has a type of the form:

\[
\exists \phi. \forall \bar{\alpha}. \tau[\bar{\alpha}] \to \exists \bar{\beta}. \sigma[\bar{\alpha}, \bar{\beta}]
\]

Compared with:

\[
\exists \phi. \forall \bar{\alpha}. \tau[\bar{\alpha}] \to \exists \bar{\beta}. \sigma[\bar{\alpha}, \bar{\beta}]
\]

That is:

- \( \sigma[\bar{\alpha}, \bar{\phi} \bar{\alpha}] \) represents the signature of the result of the functor application.
- \( \bar{\phi} \bar{\alpha} \) are the abstract types created by the application. Each \( \phi \bar{\alpha} \) is a new abstract type—one we know nothing about, as it is the application of an abstract type to \( \bar{\alpha} \).
- However, two successive applications with the same argument (hence the same \( \bar{\alpha} \)) will create two compatible structures whose signatures have the same shared abstract types \( \bar{\phi} \bar{\alpha} \).

The two applications of \( F \):

\[
\text{let } \bar{\phi}, F = \text{unpack } F \text{ in }
\]
System $F^\omega$ in OCaml

Second-order polymorphism in OCaml

• Via polymorphic methods

```ocaml
let id = object method f : \alpha. \alpha \to \alpha = fun x \to x end
let y (x : (f : \alpha. \alpha \to \alpha)) = x#f x in y id
```

• Via first-class modules

```ocaml
module type S = sig
val f : \alpha \to \alpha end
let id = (module struct let f x = x end : S)
let y (x : (module S)) = let module X = (val x) in X.f x in y id
```

Higher-order types in OCaml

• In principle, they could be encoded with first-class modules.
• Not currently possible, due to (unnecessary) restrictions.
• Modular explicits, an extension that allows a better integration of abstraction over first-class modules will remove these limitations and allow a light-weight encoding of $F^\omega$—with boiler-plate glue code.
System $F^\omega$ in OCaml

Second-order polymorphism in OCaml

- Via polymorphic methods

```ocaml
let id = object method f : \alpha. \alpha \to \alpha = fun x \to x end
let y (x : (f : \alpha. \alpha \to \alpha)) = x#f x in y id
```

- Via first-class modules

```ocaml
module type S = sig
val f : \alpha \to \alpha end
let id = (module struct
  let f x = x end : S)
let y (x : (module S)) = let module X = (val x) in X.f x in y id
```

Higher-order types in OCaml

- In principle, they could be encoded with first-class modules.
- Not currently possible, due to (unnecessary) restrictions.
- Modular explicits, an extension that allows a better integration of abstraction over first-class modules will remove these limitations and allow a light-weight encoding of $F^\omega$—with boiler-plate glue code.
System $F^\omega$ 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 $F^\omega$ in Scala-3

Higher-order polymorphism a la System $F^\omega$ is available in Scala-3. The monad example (with some variation on the signature) is:

```scala
trait Monad [F[_]] {
  def pure [A] (x: A) : F[A]
}
```

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

Still, this feature of Scala-3 is not emphasized

- It was not directly available in previous versions of Scala.
- Scala’s syntax and other complex features of Scala are obfuscating.
What’s next?

Dependent types!

Barendregt’s λ-cube

F^ω = λω → λΠω

F = λ2 → λΠ2

λ_{st} → λΠ

(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

- Introduction
- Normalization of $\lambda_{st}$
- Observational equivalence in $\lambda_{st}$
- Logical relations in stlc
- Logical relations in $F$
- Applications
- Extensions
What are logical relations?

So far, most proofs involving terms have proceeded by induction on the structure of terms (or, equivalently, on typing derivations).

Logical relations are relations between well-typed terms defined inductively on the structure of types. They allow proofs between terms by induction on the structure of types.

**Unary relations**

- Unary relations are predicates on expressions (or sets of expressions)
- They can be used to prove type safety and strong normalization

**Binary relations**

- Binary relations relate pairs of expressions of related types
- They can be used to prove equivalence of programs and non-interference properties.

*Logical relations are a common proof method for programming languages.*
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 $\forall \alpha. \alpha \to \text{int}$?

- the function cannot examine its argument
- it always returns the same integer
- $\lambda x. n$
  - $\lambda x. (\lambda y. y) \ n$
  - $\lambda x. (\lambda y. n) \ x$
  - etc.
- they are all $\beta\eta$-equivalent to the term $\lambda x. n$
Parametricity?  Inhabitants of polymorphic types

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

A term of type $\forall \alpha. \alpha \rightarrow \text{int}$?
▷ behaves as $\lambda x. n$

A term $a$ of type $\forall \alpha. \alpha \rightarrow \alpha$?
▷ behaves as $\lambda x. x$

A term type $\forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \alpha$?
▷ behaves as $\lambda x. \lambda y. x$

A term type $\forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha$?
▷ behaves either as $\lambda x. \lambda y. x$ or $\lambda x. \lambda y. y$
Similarly, the type of a polymorphic function may also reveal a “free theorem” about its behavior!

What properties may we learn from a function

\[ \text{whoami} : \forall \alpha. \text{list} \alpha \rightarrow \text{list} \alpha \]

- 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.

\[ \forall f, x, \quad \text{whoami} (\text{map} f x) = \text{map} f (\text{whoami} x) \]
Similarly, the type of a polymorphic function may also reveal a "free theorem" about its behavior!

What properties may we learn from a function

$$\text{whoami} : \forall \alpha. \text{list } \alpha \rightarrow \text{list } \alpha$$

What property may we learn for the list sorting function?

$$\text{sort} : \forall \alpha. (\alpha \rightarrow \alpha \rightarrow \text{bool}) \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha$$

If $f$ is order-preserving, then sorting commutes with $\text{map } f$

$$\forall x, y, \ \text{cmp } (f x) (f y) = \text{cmp } x y \implies \forall \ell, \ \text{sort } \text{cmp } (\text{map } f \ell) = \text{map } f (\text{sort } \text{cmp } \ell)$$
Similarly, the type of a polymorphic function may also reveal a "free theorem" about its behavior!

What properties may we learn from a function

$$\text{whoami} : \forall \alpha. \text{list} \alpha \to \text{list} \alpha$$

What property may we learn for the list sorting function?

$$\text{sort} : \forall \alpha. (\alpha \to \alpha \to \text{bool}) \to \text{list} \alpha \to \text{list} \alpha$$

If $f$ is order-preserving, then sorting commutes with $\text{map } f$

$$\left( \forall x, y, \ cmp_2 (f x) (f y) = cmp_1 x y \right) \implies \forall \ell, \ \text{sort } cmp_2 (\text{map } f \ \ell) = \text{map } f (\text{sort } cmp_1 \ \ell)$$

Application:

- If $\text{sort}$ is correct on lists of integers, then it is correct on any list
- May be useful to reduce testing.
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

\[ \text{whoami} : \forall \alpha. \text{list } \alpha \to \text{list } \alpha \]

What property may we learn for the list sorting function?

\[ \text{sort} : \forall \alpha. (\alpha \to \alpha \to \text{bool}) \to \text{list } \alpha \to \text{list } \alpha \]

If \( f \) is order-preserving, then sorting commutes with \( \text{map } f \)

\[ (\forall x, y, \ cmp_2 (f \ x) (f \ y) = cmp_1 \ x \ y) \implies \forall \ell, \ \text{sort } \ cmp_2 (\text{map } f \ \ell) = \text{map } f (\text{sort } \ cmp_1 \ \ell) \]

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

- Introduction
- **Normalization of** $\lambda_{st}$
- Observational equivalence in $\lambda_{st}$
- Logical relations in stlc
- Logical relations in F
- Applications
- Extensions
Normalization of simply-typed $\lambda$-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 $\lambda$-calculus is also lifted at the level of types in richer type systems such as $F^\omega$; 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 $\lambda$-calculus is a simple and illustrative use of logical relations.

Notice however, that our simply-typed $\lambda$-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 $\lambda$-calculus is often a difficult task because reduction may create new redexes or duplicate existing ones.

Hence the size of terms may grow (much) larger during reduction. The difficulty is to find some underlying structure that decreases.

We follow the proof schema of Pierce [2002], which is a modern presentation in a call-by-value setting of an older proof by Hindley and Seldin [1986]. The proof method is due to [Tait, 1967].
Tait’s method

Idea

- build the set $\mathcal{T}_\tau$ of terminating terms of type $\tau$;
- show that any term of type $\tau$ is in $\mathcal{T}_\tau$, by induction on terms.

This hypothesis is however too weak. The difficulty is as usual to find a strong enough induction hypothesis...

Terms of type $\tau_1 \rightarrow \tau_2$ should not only terminate but also terminate when applied to terms in $\mathcal{T}_{\tau_1}$.

The construction of $\mathcal{T}_\tau$ is thus by induction of $\tau$. 
Normalization

Definition

Let $\mathcal{T}_\tau$ be defined inductively on $\tau$ as follows:

- $\mathcal{T}_\alpha$ is the set of closed terms that terminates;
- $\mathcal{T}_{\tau_2 \to \tau_1}$ is the set of closed terms $M_1$ of type $\tau_2 \to \tau_1$ that terminates and such that $M_1 \cdot M_2$ is in $\mathcal{T}_{\tau_1}$ for any term $M_2$ in $\mathcal{T}_{\tau_2}$.

The set $\mathcal{T}_\tau$ can be seen as a predicate, i.e. a unary relation. It is called a *logical relation* because it is defined *inductively on the structure of types*. The following proofs is then schematic of the use of logical relations.
Normalization

Reduction of terms of type $\tau$ preserves membership in $\mathcal{T}_\tau$ (this is stronger than stability of $\mathcal{T}_\tau$ by reduction):

**Lemma**

\[ \text{If } \emptyset \vdash M : \tau \text{ and } M \rightarrow M', \text{ then } M \in \mathcal{T}_\tau \iff M' \in \mathcal{T}_\tau. \]

**Proof.**

The proof is by induction on $\tau$. \qed

**Lemma**

*For any type $\tau$, the reduction of any term in $\mathcal{T}_\tau$ terminates.*

Tautology, by definition of $\mathcal{T}_\tau$. 

\[ \square \]
Therefore, it just remains to show that any term of type $\tau$ is in $T_\tau$, i.e.:

**Lemma**

If $\emptyset \vdash M : \tau$, then $M \in T_\tau$.

The proof is by induction on (the typing derivation of) $M$.

However, the case for abstraction requires some similar statement, but for open terms. We need to strengthen the Lemma.

A trick to avoid considering open terms is to require the statement to hold for all closed instances of an open term:

**Lemma (strengthened)**

If $(x_i : \tau_i)^{i \in I} \vdash M : \tau$, then for any closed values $(V_i)^{i \in I}$ in $(T_{\tau_i})^{i \in I}$, the term $[(x_i \mapsto V_i)^{i \in I}] M$ is in $T_\tau$. 
Normalization

Proof. By structural induction on $M$.
We write $\Gamma$ for $(x_i : \tau_i)^{i \in I}$ and $\theta$ for $[(x_i \mapsto V_i)^{i \in I}]$. Assume $\Gamma \vdash M : \tau$.

The only interesting case is when $M$ is $\lambda x : \tau_1. M_2$:

By inversion of typing, we know that $\Gamma, x : \tau_1 \vdash M_2 : \tau_2$ and $\tau_1 \rightarrow \tau_2$ is $\tau$.

To show $\theta M \in T_\tau$, we must show that it is terminating, which holds as it is a value, and that its application to any $M_1$ in $T_{\tau_1}$ is in $T_{\tau_2}$ (1).

Let $M_1 \in T_{\tau_1}$. By definition $M_1 \rightarrow^* V$ (2). We then have:

$$(\theta M) M_1 \triangleq (\theta(\lambda x : \tau_1. M_2)) M_1$$
$$(\lambda x : \tau_1. \theta M_2) M_1$$
$$\rightarrow^* (\lambda x : \tau_1. \theta M_2) V$$
$$\rightarrow [x \mapsto V](\theta M_2)$$
$$= ([x \mapsto V] \theta)(M_2) \in T_{\tau_2} \quad \text{by induction hypothesis}$$

This establishes (1) since membership in $T_{\tau_2}$ is preserved by reduction.
Calculus

Take the call-by-value $\lambda_{st}$ with primitive booleans and conditional.

Write $B$ the type of booleans and $tt$ and $ff$ for true and false.

We define $V[T]$ and $E[T]$ the subsets of closed values and closed expressions of (ground) type $T$ by induction on types as follows:

$$V[B] \triangleq \{tt, ff\}$$

$$V[T_1 \rightarrow T_2] \triangleq \{\lambda x : T_1.M \mid \forall V \in V[T_1], (\lambda x : T_1.M) V \in E[T_2]\}$$

$$E[T] \triangleq \{M \mid \exists V \in V[T], M \Downarrow V\}$$

We write $M \Downarrow N$ for $M \rightarrow^* N$.

The goal is to show that any closed expression of type $T$ is in $E[T]$.

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. $V[T] \subseteq E[T]$—by definition. $E[T]$ is closed by inverse reduction—by definition, i.e.

If $M \Downarrow N$ and $N \in E[T]$ then $M \in E[T]$. 

1282 412
Problem

We wish to show that every closed term of type $\tau$ 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 $\emptyset \vDash M : \tau$ means $M \in \mathcal{E}[\tau]$.

We interpret free term variables of type $\tau$ as *closed values* in $\mathcal{V}[\tau]$.

We interpret environments $\Gamma$ as *closing substitutions* $\gamma$, i.e. mappings from term variables to *closed values*:

We write $\gamma \in \mathcal{G}[\Gamma]$ to mean $\text{dom}(\gamma) = \text{dom}(\Gamma)$ 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 $\emptyset \vdash M : \tau$ then $M \in \mathcal{E}[[\tau]]$.

That is, closed well-typed terms of type $\tau$ evaluate to values of type $\tau$. 
Proof by induction on the typing derivation

Routine cases

Case $\Gamma \vdash \texttt{tt} : B$ or $\Gamma \vdash \texttt{ff} : B$: by definition, $\texttt{tt}, \texttt{ff} \in \mathcal{V}[B]$ and $\mathcal{V}[B] \subseteq \mathcal{E}[B]$.

Case $\Gamma \vdash x : \tau$: $\gamma \in G[\Gamma]$, 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 $\gamma \in G[\Gamma]$. 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 $\gamma M_1 \in \mathcal{E}[\tau_2 \to \tau]$ (1) and $\gamma M_2 \in \mathcal{E}[\tau_2]$ (2).

By (2), there exists $V \in \mathcal{V}[\tau_2]$ such that $\gamma M_2 \Downarrow V$.

Thus $(\gamma M_1) (\gamma M_2) \Downarrow (\gamma M_1) V \in \mathcal{E}[\tau]$ by (1).

Then, $(\gamma M_1) (\gamma M_2) \in \mathcal{E}[\tau]$, by closure by inverse reduction.

Case $\Gamma \vdash \text{if } M \text{ then } M_1 \text{ else } M_2 : \tau$: By cases on the evaluation of $\gamma M$. 
Proof by induction on the typing derivation  

(key case)

The interesting case

Case $\Gamma \vdash \lambda x:\tau_1. M : \tau_1 \rightarrow \tau$:

Assume $\gamma \in G[\Gamma]$.
We must show that $\gamma(\lambda x:\tau_1. M) \in E[\tau_1 \rightarrow \tau]$ (1)

That is, $\lambda x:\tau_1. \gamma M \in V[\tau_1 \rightarrow \tau]$ (we may assume $x \notin \text{dom}(\gamma)$ w.l.o.g.)

Let $V \in V[\tau_1]$, it suffices to show $(\lambda x:\tau_1. \gamma M) \; V \in E[\tau]$ (2).

We have $(\lambda x:\tau_1. \gamma M) \; V \rightarrow (\gamma M)[x \mapsto V] = \gamma' M$
where $\gamma'$ is $\gamma[x \mapsto V] \in G[\Gamma, x: \tau_1]$ (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 $\gamma' M \in E[\tau]$. Since $E[\tau]$ 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:

\[ \mathcal{E}[\tau] = \{ M : \tau | \forall N, M \downarrow N \implies (N \in \mathcal{V}[\tau] \lor \exists N', N \to N') \} \]

Compare with

\[ \mathcal{E}[\tau] = \{ M : \tau | \exists V \in \mathcal{V}[\tau], M \downarrow V \} \]

Exercise

Show type soundness with this semantics.
Contents

- Introduction
- Normalization of $\lambda_{st}$
- Observational equivalence in $\lambda_{st}$
- Logical relations in stlc
- Logical relations in F
- Applications
- Extensions
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 \( \lambda x. 2 \times x \) 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 $M : B$ and $N : B$, 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)

$$
\Gamma \vdash M \simeq N : \tau \overset{\triangle}{=} \forall C : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright B), \; C[M] \simeq C[N]
$$

Typing of contexts

$$
C : (\Gamma \triangleright \tau) \rightsquigarrow (\Delta \triangleright \sigma) \iff (\forall M, \; \Gamma \vdash M : \tau \implies \Delta \vdash C[M] : \sigma)
$$

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 $M \simeq_\tau N$ for $\emptyset \vdash M \simeq N : \tau$
Observational equivalence

Observational equivalence is the coarsiest consistent congruence, where:

\[ \equiv \] is consistent if \( \emptyset \vdash M \equiv N : B \) implies \( M \simeq N \).
\[ \equiv \] is a congruence if it is an equivalence and is closed by context, i.e.
\[
\Gamma \vdash M \equiv N : \tau \land C : (\Gamma \triangleright \tau) \leadsto (\Delta \triangleright \sigma) \implies \Delta \vdash C[M] \equiv C[N] : \sigma
\]

**Consistent**: by definition, using the empty context.

**Congruence**: by compositionality of contexts.

**Coarsiest**: Assume \( \equiv \) is a consistent congruence.

We assume \( \Gamma \vdash M \equiv N : \tau \) (1) and show \( \Gamma \vdash M \simeq N : \tau \).

Let \( C : (\Gamma \triangleright \tau) \leadsto (\emptyset \triangleright B) \) (2). We must show that \( C[M] \simeq C[N] \).
This follows by consistency applied to \( \Gamma \vdash C[M] \equiv C[N] : B \) which itself follows by congruence from (1) and (2).
Problem with Observational Equivalence

Problems

- Observational equivalence is too difficult to test.
- Because of quantification over all contexts (too many for testing).
- But many contexts will do the same experiment.

Solution

We take advantage of types to reduce the number of experiments.

- Defining/testing the equivalence on base types.
- Propagating the definition mechanically at other types.

*Logical relations provide the infrastructure for conducting such proofs.*
Contents

- Introduction
- Normalization of $\lambda_{st}$
- Observational equivalence in $\lambda_{st}$
- Logical relations in stlc
- Logical relations in $F$
- Applications
- Extensions
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 $\mathcal{V}[\tau] \subseteq \text{Val}(\tau) \times \text{Val}(\tau)$.

Then, $\mathcal{E}[\tau]$ is the closure of $\mathcal{V}[\tau]$ by inverse reduction

We have $\mathcal{V}[\tau] \subseteq \mathcal{E}[\tau] \subseteq \text{Exp}(\tau) \times \text{Exp}(\tau)$. 
Logical equivalence for closed terms

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

$$\mathcal{V}[\mathsf{B}] \triangleq \{(tt, tt), (ff, ff)\}$$

$$\mathcal{V}[\tau \to \sigma] \triangleq \{(V_1, V_2) \mid V_1, V_2 \vdash \tau \to \sigma \land \forall (W_1, W_2) \in \mathcal{V}[\tau], (V_1 W_1, V_2 W_2) \in \mathcal{E}[\sigma]\}$$

$$\mathcal{E}[\tau] \triangleq \{(M_1, M_2) \mid M_1, M_2 : \tau \land \exists (V_1, V_2) \in \mathcal{V}[\tau], M_1 \downarrow V_1 \{M_2 \downarrow V_2\} M_i \downarrow V_i\}$$

where $\downarrow (M_1, M_2)$ means

In the following we will leave the typing constraint in gray implicit (as a global condition for sets $\mathcal{V}[\cdot]$ and $\mathcal{E}[\cdot]$).

We also write

$$M_1 \sim_{\tau} M_2 \text{ for } (M_1, M_2) \in \mathcal{E}[\tau] \text{ and}$$

$$V_1 \approx_{\tau} V_2 \text{ for } (V_1, V_2) \in \mathcal{V}[\tau].$$
Logical equivalence for closed terms (variant)

In a language with non-termination

We change the definition of $E[\tau]$ to

$$E[\tau] \triangleq \{(M_1, M_2) \mid M_1, M_2 : \tau \land$$

$$(\forall V_1, M_1 \Downarrow V_1 \implies \exists V_2, M_2 \Downarrow V_2 \land (V_1, V_2) \in \mathcal{V}[\tau]) \land$$

$$(\forall V_2, M_2 \Downarrow V_2 \implies \exists V_1, M_1 \Downarrow V_1 \land (V_1, V_2) \in \mathcal{V}[\tau])\}$$

Notice

$$\mathcal{V}[\tau \to \sigma] \triangleq \{(V_1, V_2) \mid V_1, V_2 \vdash \tau \to \sigma \land$$

$$\forall (W_1, W_2) \in \mathcal{V}[\tau], (V_1 W_1, V_2 W_2) \in E[\sigma]\}$$

$$= \{((\lambda x : \tau. M_1), (\lambda x : \tau. M_2)) \mid (\lambda x : \tau. M_1), (\lambda x : \tau. M_2) \vdash \tau \to \sigma \land$$

$$\forall (W_1, W_2) \in \mathcal{V}[\tau], ((\lambda x : \tau. M_1) W_1, (\lambda x : \tau. M_2) W_2) \in E[\sigma]\}$$
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 $(M_1, M_2) \in E[\tau]$, i.e. there exists $(V_1, V_2) \in \mathcal{V}[\tau]$ (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 $(N_1, M_2) \in E[\tau]$.

**Closure by inverse reduction**

Immediate, by construction of $E[\tau]$.

**Corollaries**

- If $(M_1, M_2) \in E[\tau \rightarrow \sigma]$ and $(N_1, N_2) \in E[\tau]$, then $(M_1 N_1, M_2 N_2) \in E[\sigma]$.
- To prove $(M_1, M_2) \in E[\tau \rightarrow \sigma]$, it suffices to show $(M_1 V_1, M_2 V_2) \in E[\sigma]$ for all $(V_1, V_2) \in \mathcal{V}[\tau]$. 
Properties of logical equivalence for closed terms

**Consistency** \((\sim_B) \subseteq (\simeq)\)

Immediate, by definition of \(E[B]\) and \(V[B] \subseteq (\simeq)\).

**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 \(\cong_\tau\) by induction on type \(\tau\).
Logical equivalence for closed terms

We inductively define $M_1 \sim_\tau M_2$ (read $M_1$ and $M_2$ are logically equivalent at type $\tau$) on closed terms of (ground) type $\tau$ by induction on $\tau$:

- $M_1 \sim_\land M_2$ iff $\emptyset \vdash M_1, M_2 : B$ and $M_1 \sim M_2$
- $M_1 \sim_\land M_2$ iff $\emptyset \vdash M_1, M_2 : \tau \rightarrow \sigma$ and $\forall N_1, N_2, \; N_1 \sim_\tau N_2 \implies M_1 N_1 \sim_\sigma M_2 N_2$

**Lemma**

Logical equivalence is symmetric and transitive (at any given type).

**Note**

Reflexivity is not at all obvious.
Properties of logical equivalence for closed terms (proof)

For $\sim_\tau$, the proof is immediate by transitivity and symmetry of $\approx_\tau$.

For $\approx_\tau$, it goes as follows.

*Case $\tau$ is $B$ for values:* the result is immediate.

*Case $\tau$ is $\tau \to \sigma$:*

By IH, symmetry and transitivity hold at types $\tau$ and $\sigma$.

For symmetry, assume $V_1 \approx_{\tau \to \sigma} V_2$ (H), we must show $V_2 \approx_{\tau \to \sigma} V_1$.

Assume $W_1 \approx_\tau W_2$. We must show $V_2 W_1 \sim_\sigma V_1 W_2$ (C). We have $W_2 \approx_\tau W_1$ by symmetry at type $\tau$. By (H), we have $V_2 W_2 \sim_\sigma V_1 W_1$ and (C) follows by symmetry of $\sim$ at type $\sigma$.

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

By (H1), we have $V_1 W_1 \sim_\sigma V_2 W_3$ (C1).

By symmetry and transitivity of $\approx_\tau$ (IH), we get $W_3 \approx_\tau W_3$.  

*It’s not reflexivity!*

By (H2), we have $V_2 W_3 \sim_\sigma V_3 W_3$ (C2).

(C) follows by transitivity of $\sim_\sigma$ applied to (C1) and (C2).
Logical equivalence for open terms

When $\Gamma \vdash M_1 : \tau$ and $\Gamma \vdash M_2 : \tau$, we wish to define a judgment $\Gamma \vdash M_1 \sim M_2 : \tau$ to mean that the open terms $M_1$ and $M_2$ are equivalent at type $\tau$.

The solution is to interpret program variables of $\text{dom}(\Gamma)$ by pairs of related values and typing contexts $\Gamma$ by a set of (closing) bisubstitutions $\gamma$ mapping variable type assignments to pairs of related values.

$$
\mathcal{G}[\emptyset] \triangleq \{ \emptyset \}
$$

$$
\mathcal{G}[\Gamma, x : \tau] \triangleq \{ \gamma, x \mapsto (V_1, V_2) \mid \gamma \in \mathcal{G}[\Gamma] \wedge (V_1, V_2) \in \mathcal{V}[\tau] \}
$$

Given a bisubstitution $\gamma$, we write $\gamma_i$ for the substitution that maps $x$ to $V_i$ whenever $\gamma$ 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 $\emptyset \vdash M_1 \sim M_2 : \tau$. 
Properties of logical equivalence for open terms

**Immediate properties**

Open logical equivalence is symmetric and transitive.  
(Proof is immediate by the definition and the symmetry and transitivity of closed logical equivalence.)
Fundamental lemma of logical equivalence

**Theorem (Reflexivity)** *(also called the fundamental lemma))*

*If $\Gamma \vdash M : \tau$, then $\Gamma \vdash M \sim M : \tau$.*

**Proof** By induction on the typing derivation, using compatibility lemmas.

**Compatibility lemmas**

- **C-TRUE**
  \[ \Gamma \vdash \text{tt} \sim \text{tt} : \text{bool} \]

- **C-FALSE**
  \[ \Gamma \vdash \text{ff} \sim \text{ff} : \text{bool} \]

- **C-VAR**
  \[ x : \tau \in \Gamma \quad \Gamma \vdash x \sim x : \tau \]

- **C-ABS**
  \[ \Gamma, x : \tau \vdash M_1 \sim M_2 : \sigma \quad \Gamma \vdash \lambda x : \tau. M_1 \sim \lambda x : \tau. M_2 : \tau \rightarrow \sigma \]

- **C-APP**
  \[ \Gamma \vdash M_1 \sim M_2 : \tau \rightarrow \sigma \quad \Gamma \vdash N_1 \sim N_2 : \tau \quad \Gamma \vdash M_1 N_1 \sim M_2 N_2 : \sigma \]

- **C-IF**
  \[ \Gamma \vdash M_1 \sim M_2 : \text{B} \quad \Gamma \vdash N_1 \sim N_2 : \tau \quad \Gamma \vdash N'_1 \sim N'_2 : \tau \quad \Gamma \vdash \text{if } M_1 \text{ then } N_1 \text{ else } N'_1 \sim \text{if } M_2 \text{ then } N_2 \text{ else } N'_2 : \tau \]
Proof of compatibility lemmas

Each case can be shown independently.

**Rule C-ABS**: Assume $\Gamma, x : \tau \vdash M_1 \sim M_2 : \sigma$ (1)
We show $\Gamma \vdash \lambda x : \tau. M_1 \sim \lambda x : \tau. M_2 : \tau \rightarrow \sigma$. Let $\gamma \in G[\Gamma]$.
We show $(\gamma_1(\lambda x : \tau. M_1), \gamma_2(\lambda x : \tau. M_2)) \in \mathcal{V}[\tau \rightarrow \sigma]$. Let $(V_1, V_2)$ be in $\mathcal{V}[\tau]$.
We show $(\gamma_1(\lambda x : \tau. M_1) V_1, \gamma_2(\lambda x : \tau. M_2) V_2) \in \mathcal{E}[\sigma]$ (2).

Since $\gamma_i(\lambda x : \tau. M_i) V_i \Downarrow (\gamma_i, x \mapsto V_i) M_i \triangleq \gamma'_i M_i$, by inverse reduction, it suffices to show $(\gamma'_1 M_1, \gamma'_2 M_2) \in \mathcal{E}[\sigma]$. This follows from (1) since $\gamma' \in G[\Gamma, x : \tau]$.

**Rule C-APP (and C-IF)**: By induction hypothesis and the fact that substitution distributes over applications (and conditional).

We must show $\Gamma \vdash M_1 N_1 \sim M_2 M_2 : \sigma$ (1). Let $\gamma \in G[\Gamma]$. From the premises $\Gamma \vdash M_1 \sim M_2 : \tau \rightarrow \sigma$ and $\Gamma \vdash N_1 \sim N_2 : \tau$, we have $(\gamma_1 M_1, \gamma_2 M_2) \in \mathcal{E}[\tau \rightarrow \sigma]$ 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 $\gamma \in G[\gamma]$.

We show $(\gamma_1(\text{if } M_1 \text{ then } N_1 \text{ else } N_1'), \gamma_2(\text{if } M_2 \text{ then } N_2 \text{ else } N_2')) \in E[\tau]$, That is $(\text{if } \gamma_1 M_1 \text{ then } \gamma_1 N_1 \text{ else } \gamma_1 N_1', \text{if } \gamma_2 M_2 \text{ then } \gamma_2 N_2 \text{ else } \gamma_2 N_2') \in E[\tau]$ (1).

From the premise $\Gamma \vdash M_1 \sim M_2 : B$, we have $(\gamma_1 M_1, \gamma_2 M_2) \in E[B]$. Therefore $M_1 \Downarrow V$ and $M_2 \Downarrow V$ where $V$ is either tt or ff:

- **Case $V$ is tt**. Then, $(\text{if } \gamma_i M_i \text{ then } \gamma_i N_i \text{ else } \gamma_i N_i') \Downarrow \gamma_i N_i$, 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 $(\gamma_1 N_1, \gamma_2 N_2) \in E[\tau]$ and (1) follows by closer by inverse reduction.

- **Case $V$ is ff**: similar.
Proof of reflexivity

By induction on the derivation of $\Gamma \vdash M : \tau$.
We must show $\Gamma \vdash M \sim M : \tau$:

All cases immediately follow from compatibility lemmas.

*Case $M$ is tt or ff*: Immediate by Rule $\text{C-True}$ or Rule $\text{C-False}$

*Case $M$ is $x$*: Immediate by Rule $\text{C-Var}$.

*Case $M$ is $M' N$*: By inversion of the typing rule $\text{App}$, induction hypothesis, and Rule $\text{C-App}$.

*Case $M$ is $\lambda \tau: N$*: By inversion of the typing rule $\text{Abs}$, induction hypothesis, and Rule $\text{C-Abs}$.
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 \( C : (\Gamma \triangleright \tau) \sim (\Delta \triangleright \sigma) \), then
\[
\Delta \vdash C[M] \sim C[M'] : \sigma.
\]

**Proof** By induction on the proof of \( C : (\Gamma \triangleright \tau) \sim (\Delta \triangleright \sigma) \).

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 $\tau$.

Case $B$: In the empty context, by consistency $\cong_B$ is a subrelation of $\sim_B$ which coincides with $\sim_B$.

Case $\tau \rightarrow \sigma$: By congruence of observational equivalence!

By hypothesis, we have $M_1 \cong_{\tau \rightarrow \sigma} M_2$ (1). To show $M_1 \sim_{\tau \rightarrow \sigma} M_2$, we assume $V_1 \sim_\tau V_2$ (2) and show $M_1 V_1 \sim_\sigma M_2 V_2$ (3).

By soundness applied to (2), we have $V_1 \cong_\tau V_2$ from (2). By congruence with (1), we have $M_1 V_1 \cong_\sigma M_2 V_2$, which implies (3) by IH at type $\sigma$. 

\begin{align*}
&\text{\textbf{Completeness of logical equivalence}} \\
&\text{Observational equivalence of closed terms implies logical equivalence. That is } (\cong_\tau) \subseteq (\sim_\tau). \\
&\text{Proof by induction on } \tau. \\
&\text{Case } B: \text{ In the empty context, by consistency } \cong_B \text{ is a subrelation of } \sim_B \text{ which coincides with } \sim_B. \\
&\text{Case } \tau \rightarrow \sigma: \text{ By congruence of observational equivalence!} \\
&\text{By hypothesis, we have } M_1 \cong_{\tau \rightarrow \sigma} M_2 \text{ (1). To show } M_1 \sim_{\tau \rightarrow \sigma} M_2, \text{ we assume } V_1 \sim_\tau V_2 \text{ (2) and show } M_1 V_1 \sim_\sigma M_2 V_2 \text{ (3).} \\
&\text{By soundness applied to (2), we have } V_1 \cong_\tau V_2 \text{ from (2). By congruence with (1), we have } M_1 V_1 \cong_\sigma M_2 V_2, \text{ which implies (3) by IH at type } \sigma.
\end{align*}
**Logical equivalence: example of application**

**Fact:** Assume $\textit{not} \triangleq \lambda x:B.\text{if } x \text{ then } \texttt{ff} \text{ else } \texttt{tt}$

and $M \triangleq \lambda x:B.\lambda y:\tau.\lambda z:\tau.\text{if } \textit{not} x \text{ then } y \text{ else } z$

and $M' \triangleq \lambda x:B.\lambda y:\tau.\lambda z:\tau.\text{if } x \text{ then } z \text{ else } y$.

Show that $M \cong_{B\rightarrow\tau\rightarrow\tau\rightarrow\tau} M'$.

**Proof**

It suffices to show $M \ V_0 \ V_1 \ V_2 \sim_\tau M' \ V_0' \ V_1' \ V_2'$ whenever $V_0 \approx_B V_0'$ (1) and $V_1 \approx_\tau V_1'$ (2) and $V_2 \approx_\tau V_2'$ (3). By inverse reduction, it suffices to show: if $\textit{not} \ V_0$ then $V_1$ else $V_2 \sim_\tau$ if $V_0'$ then $V_2'$ else $V_1'$ (4).

It follows from (1) that we have only two cases:

**Case** $V_0 = V_0' = \texttt{tt}$: Then $\textit{not} \ V_0 \Downarrow \texttt{ff}$ and thus $M \Downarrow V_2$ while $M' \Downarrow V_2$. Then (4) follows by inverse reduction and (3).

**Case** $V_0 = V_0' = \texttt{ff}$: is symmetric.
Contents

- Introduction
- Normalization of $\lambda_{st}$
- Observational equivalence in $\lambda_{st}$
- Logical relations in stlc
- Logical relations in F
- Applications
- Extensions
Observational equivalence

We now extend the notion of logical equivalence to System F.

\[ \tau ::= \ldots | \alpha | \forall \alpha. \tau \]

\[ M ::= \ldots | \Lambda \alpha. M | M \tau \]

We write typing contexts \( \Delta; \Gamma \) where \( \Delta \) binds variables and \( \Gamma \) binds program variables.

Typing of contexts becomes \( C : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\Delta'; \Gamma' \triangleright \tau') \).

**Observational equivalence**

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

\[ \forall C : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\emptyset; \emptyset \triangleright B), \quad C[M] \simeq C[M'] \]

As before, write \( M \equiv_{\tau} N \) for \( \emptyset; \emptyset \vdash M \equiv N : \tau \) (in particular, \( \tau \) is closed).
Logical equivalence

For closed terms (no free program variables)

- **We need to give the semantics of polymorphic 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 $\rho_1$ and $\rho_2$

Let $R(\rho_1, \rho_2)$ be the set of relations on values of closed types $\rho_1$ and $\rho_2$, that is $\mathcal{P}(\text{Val}(\rho_1) \times \text{Val}(\rho_2))$. We optionally restrict to *admissible* relations, i.e. relations that are *closed by observational equivalence*:

$$R \in R^\#(\tau_1, \tau_2) \implies \forall (V_1, V_2) \in R, \forall W_1, W_2, W_1 \simeq V_1 \land W_2 \simeq V_2 \implies (W_1, W_2) \in R$$

The restriction to *admissible relations* is required for *completeness* of logical equivalence with respect to observational equivalence but *not for soundness*. 
Example of admissible relations

For example, both

\[
R_1 \overset{\Delta}{=} \{(tt, 0), (ff, 1)\}
\]
\[
R_2 \overset{\Delta}{=} \{(tt, 0)\} \cup \{(ff, n) \mid n \in \mathbb{Z}^*\}
\]

are admissible relations in \(\mathcal{R}^\#(B, \text{int})\).

But

\[
R_3 \overset{\Delta}{=} \{(tt, \lambda x: \tau. 0), (ff, \lambda x: \tau. 1)\}
\]

although in \(\mathcal{R}(B, \tau \to \text{int})\), is not admissible.

Taking \(M_0 \overset{\Delta}{=} \lambda x: \tau. (\lambda z: \text{int}. z) 0\), we have \(M \equiv_{\tau \to \text{int}} \lambda x: \tau. 0\) but \((tt, M)\) is not in \(R_3\). Note A relation \(R\) in \(\mathcal{R}(\tau_1, \tau_2)\) can always be turned into an admissible relation \(R^\#\) in \(\mathcal{R}^\#(\tau_1, \tau_2)\) by closing \(R\) by observational equivalence.

Note It is a key that such relations can relate values at different types.
Interpretation of type environments

Interpretation of type variables

We write $\eta$ for mappings $\alpha \mapsto (\rho_1, \rho_2, R)$ where $R \in \mathcal{R}(\rho_1, \rho_2)$.

We write $\eta_i$ (resp. $\eta_R$) for the type (resp. relational) substitution that maps $\alpha$ to $\rho_i$ (resp. $R$) whenever $\eta$ maps $\alpha$ to $(\rho_1, \rho_2, R)$.

We define

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

$$\mathcal{V}[\forall \alpha. \tau]_\eta \triangleq \{(V_1, V_2) \mid V_1 : \eta_1(\forall \alpha. \tau) \land V_2 : \eta_2(\forall \alpha. \tau) \land$$

$$\land \forall \rho_1, \rho_2, \forall R \in \mathcal{R}(\rho_1, \rho_2), (V_1 \rho_1, V_2 \rho_2) \in \mathcal{E}[\tau]_{\eta, \alpha \mapsto (\rho_1, \rho_2, R)}\}$$
Logical equivalence for closed terms with open types

We redefine

\[ \mathcal{V}[B]_{\eta} \triangleq \{(tt, tt), (ff, ff)\} \]

\[ \mathcal{V}[\tau \rightarrow \sigma]_{\eta} \triangleq \{(V_1, V_2) \mid V_1 \vdash \eta_1(\tau \rightarrow \sigma) \land V_2 \vdash \eta_2(\tau \rightarrow \sigma) \land \forall (W_1, W_2) \in \mathcal{V}[\tau]_{\eta}, (V_1 W_1, V_2 W_2) \in \mathcal{E}[\sigma]_{\eta}\} \]

\[ \mathcal{E}[\tau]_{\eta} \triangleq \{(M_1, M_2) \mid M_1 : \eta_1 \tau \land M_2 : \eta_2 \tau \land \exists (V_1, V_2) \in \mathcal{V}[\tau]_{\eta}, M_1 \downarrow V_1 \land M_2 \downarrow V_2\} \]

\[ \mathcal{G}[\emptyset]_{\eta} \triangleq \{\emptyset\} \]

\[ \mathcal{G}[\Gamma, x : \tau]_{\eta} \triangleq \{\gamma, x \mapsto (V_1, V_2) \mid \gamma \in \mathcal{G}[\Gamma]_{\eta} \land (V_1, V_2) \in \mathcal{V}[\tau]_{\eta}\} \]

and define

\[ \mathcal{D}[\emptyset] \triangleq \{\emptyset\} \]

\[ \mathcal{D}[\Delta, \alpha] \triangleq \{\eta, \alpha \mapsto (\rho_1, \rho_2, R) \mid \eta \in \mathcal{D}[\Delta] \land R \in \mathcal{R}(\rho_1, \rho_2)\} \]
Logical equivalence for open terms

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

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

*(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. $\emptyset; \emptyset \vdash M_1 \sim M_2 : \tau$).*

In this case, $\tau$ is a closed type and $M_1$ and $M_2$ are closed terms of type $\tau$; hence, this coincides with the previous definition $(M_1, M_2)$ in $E[\tau]$, which may still be used as a shorthand for $E[\tau]_{\emptyset}$.
Properties

Respect for observational equivalence

If \((M_1, M_2) \in \mathcal{E}[[\tau]]^\#_\eta \) and \(N_1 \cong_{\eta_1(\tau)} M_1\) and \(N_2 \cong_{\eta_2(\tau)} M_2\) then \((N_1, N_2) \in \mathcal{E}[[\tau]]^\#_\eta\).

(We use \(\#\) to indicate that admissibility is required in the definition of \(\mathcal{R}^\#\))

Proof. By induction on \(\tau\).

Assume \((M_1, M_2) \in \mathcal{E}[[\tau]]_\eta\) (1) and \(N_1 \cong_{\eta_1(\tau)} M_1\) (2). We show \((N_1, M_2) \in \mathcal{E}[[\tau]]_\eta\).

Case \(\tau\) is \(\forall \alpha. \sigma\): Assume \(R \in \mathcal{R}^\# (\rho_1, \rho_2)\). Let \(\eta_\alpha\) 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 \(\tau\) is \(\alpha\): Relies on admissibility, indeed.

Other cases: the proof is similar to the case of the simply-typed \(\lambda\)-calculus.

Corollary
Properties

Lemma (Closure under observational equivalence)
If $\Delta; \Gamma \vdash M_1 \sim^\# M_2 : \tau$ and $\Delta; \Gamma \vdash M_1 \cong N_1 : \tau$ and $\Delta; \Gamma \vdash M_2 \cong N_2 : \tau$,
then $\Delta; \Gamma \vdash N_1 \sim^\# N_2 : \tau$

Requirements: admissibility

Lemma (Compositionality)

Assume $\Delta \vdash \sigma$ and $\Delta, \alpha \vdash \tau$ and $\eta \in D[\Delta]$. Then,

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

Proof by induction on $\tau$. Key lemma
Parametricity

Theorem (Reflexivity) (also called the fundamental lemma)

*If* \( \Delta; \Gamma \vdash M : \tau \) *then* \( \Delta; \Gamma \vdash M \sim M : \tau \).  

**Notice:** Admissibility is not required for the fundamental lemma.

**Proof** by induction on the typing derivation, using compatibility lemmas.

**Compatibility lemmas**

We redefine the lemmas to work in a typing context of the form \( \Delta, \Gamma \) instead of \( \Gamma \) and add two new lemmas:

\[
\begin{align*}
\text{C-TABS} & \quad \Delta, \alpha; \Gamma \vdash M_1 \sim M_2 : \tau \\
\hline
\Delta; \Gamma \vdash \Lambda \alpha. M_1 \sim \Lambda \alpha. M_2 : \forall \alpha. \tau
\end{align*}
\]

\[
\begin{align*}
\text{C-TAPP} & \quad \Delta; \Gamma \vdash M_1 \sim M_2 : \forall \alpha. \tau \\
\hline
\Delta; \Gamma \vdash M_1 \sigma \sim M_2 \sigma : \tau[\alpha \mapsto \sigma]
\end{align*}
\]
Proof of compatibility

Case $M$ is $\Lambda\alpha. N$: We must show that $\Delta; \Gamma \vdash \Lambda\alpha. N \sim \Lambda\alpha. N : \forall\alpha. \tau$.
Assume $\eta : \delta \leftrightarrow_\Delta \delta'$ and $\gamma \sim_\Gamma \gamma' [\eta : \delta \leftrightarrow \delta']$.

We must show $\gamma(\delta(\Lambda\alpha. N)) \sim_{\forall\alpha. \tau} \gamma'(\delta(\Lambda\alpha. N)) [\eta : \delta \leftrightarrow \delta]$.

Assume $\sigma$ and $\sigma'$ closed and $R : \sigma \leftrightarrow \sigma'$. We must show

$$(\gamma(\delta(\Lambda\alpha. N))) \sigma \sim_\tau (\gamma'(\delta'(\Lambda\alpha. N))) \sigma [\eta_0 : \delta_0 \leftrightarrow \delta'_0]$$

where $\eta_0 = \eta, \alpha \mapsto R$ and $\delta_0 = \delta, \alpha \mapsto \sigma$ and $\delta'_0 = \delta, \alpha \mapsto \sigma'$.

Since

$$(\gamma(\delta(\Lambda\alpha. N))) \sigma = (\Lambda\alpha. \gamma(\delta(N))) \sigma \rightarrow \gamma(\delta(N))[\alpha \mapsto \sigma] = \gamma(\delta_0(N))$$

It suffices to show

$$\gamma(\delta_0(N)) \sim_\tau \gamma'(\delta'_0(N)) [\eta_0 : \delta_0 \leftrightarrow \delta'_0]$$

which follows by IH from $\Delta, \alpha; \Gamma \vdash N : \tau$ (which we obtain from $\Delta, \Gamma \vdash \Lambda\alpha. N : \tau$ by inversion).
Proof of compatibility

Case $M$ is $N \sigma$:

By inversion of typing $\Delta, \Gamma \vdash N : \forall \alpha. \tau_0$ (1) and $\tau$ is $\forall \alpha. \tau_0$.
We must show that $\Delta; \Gamma \vdash N \sigma \sim N \sigma : \tau_0[\alpha \mapsto \sigma]$.

Assume $\eta : \delta \leftrightarrow \Delta \delta'$ and $\gamma \sim_{\Gamma} \gamma'$ $[\eta : \delta \leftrightarrow \delta']$. We must show

$$
\gamma(\delta(N \sigma)) \sim_{\tau_0[\alpha \mapsto \sigma]} \gamma'(\delta'(N \sigma))$ [\eta : \delta \leftrightarrow \delta']

i.e. $$(\gamma(\delta(N))) \sigma \sim_{\tau_0[\alpha \mapsto \sigma]} (\gamma'(\delta'(N))) \sigma$ [\eta : \delta \leftrightarrow \delta']$$

By compositionality, it suffices to show

$$(\gamma(\delta(N))) \sigma \sim_{\tau_0} (\gamma'(\delta'(N))) \sigma$$ [\eta_0 : \delta_0 \leftrightarrow \delta'_0] \quad (2)

where $\eta_0 = \eta, \alpha \mapsto R$ and $\delta_0 = \delta, \alpha \mapsto \sigma$ and $\delta'_0 = \delta, \alpha \mapsto \sigma'$ and $R : \delta(s) \leftrightarrow \delta'(s)$ is defined by $R(N_0, N'_0) \iff N_0 \sim_{\sigma} N'_0$ [\eta : \delta \leftrightarrow \delta'].

This relation is admissible (3). Hence by IH from (1), we have

$$(\gamma(\delta(N))) \sim_{\forall \alpha. \tau_0} (\gamma'(\delta'(N)))$$ $[\eta : \delta \leftrightarrow \delta']$$

which implies (2) by definition of $\sim_{\forall \alpha. \tau_0}$.
Properties

**Soundness of logical equivalence**
Logical equivalence implies observational equivalence.
If $\Delta; \Gamma \vdash M_1 \sim M_2 : \tau$ then $\Delta; \Gamma \vdash M_1 \cong M_2 : \tau$.

**Completeness of logical equivalence**
Observational equivalence implies logical equivalence with admissibility.
If $\Delta; \Gamma \vdash M_1 \cong M_2 : \tau$ then $\Delta; \Gamma \vdash M_1 \sim^\# M_2 : \tau$.

As a particular case, $M_1 \cong^\tau M_2$ iff $M_1 \sim^\#^\tau M_2$.

**Note:** Admissibility is not required for soundness—only for completeness.
That is, proofs that some observational equivalence hold do not usually require admissibility.
Properties

Extensionality

(A fact, hence does not depend on admissibility)

\[ M_1 \cong_{\tau \rightarrow \sigma} M_2 \text{ iff } \forall (V : \tau), M_1 V \cong_{\sigma} M_2 V \text{ iff } \forall (N : \tau), M_1 N \cong_{\sigma} M_2 N \]

\[ M_1 \cong_{\forall \alpha. \tau} M_2 \text{ iff for all closed type } \rho, M_1 \rho \cong_{\tau[\alpha \mapsto \rho]} M_2 \rho. \]

Proof. Forward direction is immediate as \( \cong \) is a congruence. Backward direction uses logical relations and admissibility, but the exported statement does not.

Case Value abstraction: It suffices to show \( M_1 \sim_{\tau \rightarrow \sigma} M_2 \). That is, assuming \( N_1 \sim_{\tau} N_2 \) (1), we show \( M_1 N_1 \sim_{\sigma} M_2 N_2 \) (2). By assumption, we have \( M_1 N_1 \cong_{\sigma} M_2 N_1 \) (3). By the fundamental lemma, we have \( M_2 \sim_{\tau \rightarrow \sigma} M_2 \). Hence, from (1), we must have \( M_2 N_1 \sim_{\sigma} M_2 N_2 \). We conclude (2) by respect for observational equivalence with (3)—which requires admissibility.

Case Type abstraction: It suffices to show \( M_1 \sim_{\forall \alpha. \tau} M_2 \). That is, given \( R \in \mathcal{R}(\rho_1, \rho_2) \), we show \( (M_1 \rho_1, M_2 \rho_2) \in \mathcal{E}[\tau]_{\alpha \mapsto (\rho_1, \rho_2, R)} \) (4). By assumption, we have \( M_1 \rho_1 \cong_{\tau[\alpha \mapsto \rho_1]} M_2 \rho_1 \) (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

Let $\theta$ be a substitution of type variables for ground types. Let $R$ be the restriction of $\simeq_{\alpha\theta}$ to $\text{Val}(\alpha\theta) \times \text{Val}(\alpha\theta)$ and $\eta : \alpha \mapsto (\alpha\theta, \alpha\theta, R)$.

Then $E[\tau]_{\eta}$ is equal to $\simeq_{\tau\theta}$.

(The proof uses respect for observational equivalence, which requires admissibility)
Contents

- Introduction
- Normalization of $\lambda_{st}$
- Observational equivalence in $\lambda_{st}$
- Logical relations in stlc
- Logical relations in F
- Applications
- Extensions
Applications

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

**Fact** If $M : \forall \alpha. \alpha \to \alpha$, then $M \equiv_{\forall \alpha. \alpha \to \alpha} \text{id}$ where $\text{id} \triangleq \Lambda \alpha. \lambda x : \alpha. x$.

**Proof** By *extensionality*, it suffices to show that for any $\rho$ and $V : \rho$ we have $M \rho V \equiv_{\rho} \text{id} \rho V$. In fact, by closure by inverse reduction, it suffices to show $M \rho V \equiv_{\rho} V$ (1).

By parametricity, we have $M \sim_{\forall \alpha. \alpha \to \alpha} M$ (2).

Consider $R$ in $\mathcal{R}(\rho, \rho)$ equal to $\{(V, V)\}$ and $\eta$ be $[\alpha \mapsto (\rho, \rho, R)]$. (3)

By construction, we have $(V, V) \in \mathcal{V}[\alpha]_{\eta}$.

Hence, from (2), we have $(M \rho V, M \rho V) \in \mathcal{E}[\alpha]_{\eta}$, which means that the pair $(M \rho V, M \rho V)$ reduces to a pair of values in (the singleton) $R$. This implies that $M \rho V$ reduces to $V$, which in turn, implies (1).

(3) Admissibility is not needed
Applications

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

**Fact** Let $\sigma$ be $\forall \alpha. \alpha \to \alpha \to \alpha$. If $M : \sigma$, then either

$M \equiv_\sigma W_1 \overset{\Delta}{=} \Lambda \alpha. \lambda x_1 : \alpha. \lambda x_2 : \alpha. x_1$ or $M \equiv_\sigma W_2 \overset{\Delta}{=} \Lambda \alpha. \lambda x_1 : \alpha. \lambda x_2 : \alpha. x_2$

**Proof** By **extensionality**, it suffices to show that for either $i = 1$ or $i = 2$, for any closed type $\rho$ and $V_1, V_2 : \rho$, we have $M \rho V_1 V_2 \equiv_\rho W_i \rho V_1 V_2$, or just $M \rho V_1 V_2 \equiv_\sigma V_i \ (1)$.

Let $\rho$ and $V_1, V_2 : \rho$ be fixed. Consider $R$ equal to $\{(tt, V_1), (ff, V_2)\}$ in $\mathcal{R}(B, \rho)$ and $\eta$ be $\alpha \mapsto (B, \rho, R)$. We have $(tt, V_1) \in \mathcal{V}[\alpha]\eta$ since $R(tt, V_1)$ and, similarly, $(ff, V_2) \in \mathcal{V}[\alpha]\eta$.

We have $(M, M) \in \mathcal{E}[\sigma]$ by parametricity. Hence, $(M B \ tt \ ff, M \rho V_1 V_2)$ is in $\mathcal{V}[\alpha]\eta$, which means that $(M B \ tt \ ff, M \rho V_1 V_2)$ reduces to a pair of values in $R$, which implies:

$$\forall \rho, V_1, V_2, \bigvee \begin{cases} \forall \rho, V_1, V_2, \ M B \ tt \ ff \equiv_B tt \land M \rho V_1 V_2 \equiv_\rho V_1 \\
\forall \rho, V_1, V_2, \ M B \ tt \ ff \equiv_B ff \land M \rho V_1 V_2 \equiv_\rho V_2 \end{cases}$$

Since, $M B \ tt \ ff$ is independent of $\rho, V_1, and V_2$, this actually shows (1).
Applications

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

Fact Let $\sigma$ be $\forall \alpha. \alpha \to \alpha \to \alpha$. If $M : \sigma$, then either $M \simeq_{\sigma} W_1 \triangleq \Lambda \alpha. \lambda x_1 : \alpha. \lambda x_2 : \alpha. x_1$ or $M \simeq_{\sigma} W_2 \triangleq \Lambda \alpha. \lambda x_1 : \alpha. \lambda x_2 : \alpha. x_2$.

Proof By extensionality, it suffices to show that for either $i = 1$ or $i = 2$, for any closed type $\rho$ and $V_1, V_2 : \rho$, we have $M \rho V_1 V_2 \simeq_{\rho} W_i \rho V_1 V_2$, or just $M \rho V_1 V_2 \simeq_{\sigma} V_i$ (1).

Let $\rho$ and $V_1, V_2 : \rho$ be fixed. Consider $R$ equal to $\{(tt, V_1), (ff, V_2)\}$ in $R(B, \rho)$ and $\eta$ be $\alpha \mapsto (B, \rho, R)$. We have $(tt, V_1) \in \mathcal{V}[\alpha]_{\eta}$ since $R(tt, V_1)$ and, similarly, $(ff, V_2) \in \mathcal{V}[\alpha]_{\eta}$.

We have $(M, M) \in \mathcal{E}[\sigma]$ by parametricity. Hence, $(M \ B \ tt \ ff, M \rho V_1 V_2)$ is in $\mathcal{V}[\alpha]_{\eta}$, which means that $(M \ B \ tt \ ff, M \rho V_1 V_2)$ reduces to a pair of values in $R$, which implies:

$$\forall \rho, V_1, V_2, \bigvee \left\{ \begin{array}{ll}
\forall \rho, V_1, V_2, & M \ B \ tt \ ff \simeq_B \ tt \ \land \ M \rho V_1 V_2 \simeq_{\rho} V_1 \\
\forall \rho, V_1, V_2, & M \ B \ tt \ ff \simeq_B \ ff \ \land \ M \rho V_1 V_2 \simeq_{\rho} V_2 \end{array} \right. $$

Since, $M \ B \ tt \ ff$ is independent of $\rho, V_1,$ and $V_2$, this actually shows (1).
Applications

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

Fact Let $\sigma$ be $\forall \alpha. \alpha \to \alpha \to \alpha$. If $M : \sigma$, then either

$M \equiv_{\sigma} W_1 \overset{\Delta}{=} \Lambda \alpha. \lambda x_1 : \alpha. \lambda x_2 : \alpha. x_1$ or $M \equiv_{\sigma} W_2 \overset{\Delta}{=} \Lambda \alpha. \lambda x_1 : \alpha. \lambda x_2 : \alpha. x_2$

Proof By extensionality, it suffices to show that for either $i = 1$ or $i = 2$, for any closed type $\rho$ and $V_1, V_2 : \rho$, we have $M \rho V_1 V_2 \equiv_{\rho} W_i \rho V_1 V_2$, or just $M \rho V_1 V_2 \equiv_{\sigma} V_i$ (1).

Let $\rho$ and $V_1, V_2 : \rho$ be fixed. Consider $R$ equal to $\{(tt, V_1), (ff, V_2)\}$ in $\mathcal{R}(B, \rho)$ and $\eta$ be $\alpha \mapsto (B, \rho, R)$. We have $(tt, V_1) \in \mathcal{V}[\alpha]_{\eta}$ since $R(tt, V_1)$ and, similarly, $(ff, V_2) \in \mathcal{V}[\alpha]_{\eta}$.

We have $(M, M) \in \mathcal{E}[\sigma]$ by parametricity. Hence, $(M B tt ff, M \rho V_1 V_2)$ is in $\mathcal{V}[\alpha]_{\eta}$, which means that $(M B tt ff, M \rho V_1 V_2)$ reduces to a pair of values in $R$, which implies:

$$\forall \rho, V_1, V_2, \bigvee \left\{ \begin{align*}
\forall \rho, V_1, V_2, & M B tt ff \equiv_B tt \land M \rho V_1 V_2 \equiv_{\rho} V_1 \\
\forall \rho, V_1, V_2, & M B tt ff \equiv_B ff \land M \rho V_1 V_2 \equiv_{\rho} V_2
\end{align*} \right\}$$

Since, $M B tt ff$ is independent of $\rho$, $V_1$, and $V_2$, this actually shows (1).
Applications

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

Fact Let $\sigma$ be $\forall \alpha. \alpha \to \alpha \to \alpha$. If $M : \sigma$, then either $M \simeq_{\sigma} W_1 \triangleq \Lambda \alpha. \lambda x_1: \alpha. \lambda x_2: \alpha. x_1$ or $M \simeq_{\sigma} W_2 \triangleq \Lambda \alpha. \lambda x_1: \alpha. \lambda x_2: \alpha. x_2$

Proof By extensionality, it suffices to show that for either $i = 1$ or $i = 2$, for any closed type $\rho$ and $V_1, V_2 : \rho$, we have $M \rho V_1 V_2 \simeq_{\rho} W_i \rho V_1 V_2$, or just $M \rho V_1 V_2 \simeq_{\sigma} V_i \ (1)$.

Let $\rho$ and $V_1, V_2 : \rho$ be fixed. Consider $R$ equal to $\{(tt, V_1), (ff, V_2)\}$ in $R(B, \rho)$ and $\eta$ be $\alpha \mapsto (B, \rho, R)$. We have $(tt, V_1) \in \mathcal{V}[\alpha]_\eta$ since $R(tt, V_1)$ and, similarly, $(ff, V_2) \in \mathcal{V}[\alpha]_\eta$.

We have $(M, M) \in E[\sigma]$ by parametricity. Hence, $(M B \ \tt \ ff, M \rho V_1 V_2)$ is in $\mathcal{V}[\alpha]_\eta$, which means that $(M B \ tt \ ff, M \rho V_1 V_2)$ reduces to a pair of values in $R$, which implies:

$$\forall \rho, V_1, V_2, \ \vee \begin{cases} \forall \rho, V_1, V_2, \ M B \ tt \ ff \ \simeq_B \ tt \ \land \ M \rho V_1 V_2 \ \simeq_{\rho} V_1 \\ \forall \rho, V_1, V_2, \ M B \ tt \ ff \ \simeq_B \ ff \ \land \ M \rho V_1 V_2 \ \simeq_{\rho} V_2 \end{cases}$$

Since, $M B \ tt \ ff$ is independent of $\rho, V_1, \text{ and } V_2$, this actually shows $(1)$.\[\]
Applications

Inhabitants of $\forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha$

**Fact** Let $\sigma$ be $\forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha$. 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 $\rho$ 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 $\rho$ and $V_1, V_2 : \rho$ be fixed. Consider $R$ equal to $\{(0, V_1), (1, V_2)\}$ in $R(\mathbb{N}, \rho)$ and $\eta$ be $\alpha \mapsto (\mathbb{N}, \rho, R)$. We have $(0, V_1) \in \mathcal{V}[\alpha]_\eta$ since $R(0, V_1)$ and, similarly, $(1, V_2) \in \mathcal{V}[\alpha]_\eta$.

We have $(M, M) \in \mathcal{E}[\sigma]$ by parametricity. Hence, $(M \mathbb{N} \ 0 \ 1 \ 1, M \rho V_1 V_2)$ is in $\mathcal{V}[\alpha]_\eta$, which means that $(M \mathbb{N} \ 0 \ 1 \ 1, M \rho V_1 V_2)$ reduces to a pair of values in $R$, which implies:

$$\forall \rho, V_1, V_2, \bigvee \left\{ \begin{align*}
\forall \rho, V_1, V_2, & \quad M \mathbb{N} \ 0 \ 1 \ \cong_{\mathbb{N}} \ 0 \ \wedge \ M \rho V_1 V_2 \cong_{\rho} V_1 \\
\forall \rho, V_1, V_2, & \quad M \mathbb{N} \ 0 \ 1 \ \cong_{\mathbb{N}} \ 1 \ \wedge \ M \rho V_1 V_2 \cong_{\rho} V_2
\end{align*} \right\}$$

*Since, $M \mathbb{N} \ 0 \ 1 \ 1$ is independent of $\rho, V_1,$ and $V_2$, this actually shows (1).*
Applications

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

**Fact** Let $\sigma$ be $\forall \alpha. \alpha \to \alpha \to \alpha$. If $M : \sigma$, then either

$$M \equiv_\sigma W_1 \triangleq \Lambda \alpha. \lambda x_1 : \alpha. \lambda x_2 : \alpha. x_1 \quad \text{or} \quad M \equiv_\sigma W_2 \triangleq \Lambda \alpha. \lambda x_1 : \alpha. \lambda x_2 : \alpha. x_2$$

**Proof** By *extensionality*, it suffices to show that for either $i = 1$ or $i = 2$, for any closed type $\rho$ and $V_1, V_2 : \rho$, we have $M \rho V_1 V_2 \equiv_\rho W_i \rho V_1 V_2$, or just $M \rho V_1 V_2 \equiv_\sigma V_i$ (1).

Let $\rho$ and $V_1, V_2 : \rho$ be fixed. Consider $R$ equal to $\{ (W_1, V_1), (W_2, V_2) \}$ in $R(\sigma, \rho)$ and $\eta$ be $\alpha \mapsto (\sigma, \rho, R)$. We have $(W_1, V_1) \in \mathcal{V}[\alpha]_\eta$ since $R(W_1, V_1)$ and, similarly, $(W_2, V_2) \in \mathcal{V}[\alpha]_\eta$.

We have $(M, M) \in \mathcal{E}[\sigma]$ by parametricity. Hence, $(M \sigma W_1 W_2, M \rho V_1 V_2)$ is in $\mathcal{V}[\alpha]_\eta$, which means that $(M \sigma W_1 W_2, M \rho V_1 V_2)$ reduces to a pair of values in $R$, which implies:

$$\forall \rho, V_1, V_2, \quad \bigvee \left\{ \begin{array}{l}
\forall \rho, V_1, V_2, \quad M \sigma W_1 W_2 \equiv_\sigma W_1 \quad \land \quad M \rho V_1 V_2 \equiv_\rho V_1 \\
\forall \rho, V_1, V_2, \quad M \sigma W_1 W_2 \equiv_\sigma W_2 \quad \land \quad M \rho V_1 V_2 \equiv_\rho V_2
\end{array} \right\}$$

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

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

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

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

**Fact** Let $nat$ be $\forall \alpha. (\alpha \to \alpha) \to \alpha \to \alpha$. If $M : nat$, then $M \cong_{nat} N_n$ for some integer $n$, where $N_n \overset{\Delta}{=} \Lambda \alpha. \lambda f : \alpha \to \alpha. \lambda x : \alpha. f^n x$. 
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 $M : nat$, then $M \cong_{nat} N_n$ for some integer $n$, where $N_n \triangleq \Lambda \alpha. \lambda f : \alpha \to \alpha. \lambda x : \alpha. f^n x$. 
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 $N_n \triangleq \Lambda \alpha. \lambda f : \alpha \to \alpha. \lambda x : \alpha. f^n x$.

That is, the inhabitants of $\forall \alpha. (\alpha \to \alpha) \to \alpha \to \alpha$ are the Church naturals.

Proof By extensionality, it suffices to show that there exists $n$ such that for any closed type $\rho$ 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, $M \rho V_1 V_2 \sim_{\rho} V_1^n V_2$ (1), since $N_n \rho V_1 V_2$ reduces to $V_1^n V_2$. Let $\rho$ 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 $R(nat, \rho)$ be $\{(S^k Z, V_1^k V_2) \mid k \in \mathbb{N}\}$ and $\eta$ be $\alpha \mapsto (nat, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\alpha]_\eta$.

We also have $(S, V_1) \in \mathcal{V}[\alpha \to \alpha]_\eta$. (A key to the proof.)
Applications

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

**Fact** Let $nat$ be $\forall \alpha. (\alpha \to \alpha) \to \alpha \to \alpha$. If $M : nat$, 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 $\rho$ 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, $M \rho V_1 V_2 \sim_{\rho} V_1^n V_2$ (1), since $N_n \rho V_1 V_2$ reduces to $V_1^n V_2$. Let $\rho$ 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 $\mathcal{R}(nat, \rho)$ be $\{ (S^k Z, V_1^k V_2) \mid k \in \mathbb{N} \}$ and $\eta$ be $\alpha \mapsto (nat, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\alpha]_{\eta}$.

We also have $(S, V_1) \in \mathcal{V}[\alpha \to \alpha]_{\eta}$.

(A key to the proof.)
Applications

Fact Let \( \text{nat} \) be \( \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \). If \( M : \text{nat} \), then \( M \cong_{\text{nat}} N_n \) for some integer \( n \), where \( N_n \triangleq \Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. \lambda x : \alpha. f^n x \).

Proof By \textit{extensionality}, it suffices to show that there exists \( n \) such that for any closed type \( \rho \) and closed values \( V_1 : \rho \rightarrow \rho \) 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, \( M \rho V_1 V_2 \leadsto_{\rho} V_1^n V_2 \) (1), since \( N_n \rho V_1 V_2 \) reduces to \( V_1^n V_2 \). Let \( \rho \) and \( V_1 : \rho \rightarrow \rho \) and \( V_2 : \rho \) be fixed.

Let \( Z \) be \( N_0 \text{nat} \) and \( S \) be \( N_1 \text{nat} \). Let \( R \) in \( \mathcal{R}(\text{nat}, \rho) \) be \( \{ (S^k Z, V_1^k V_2) \mid k \in \mathbb{N} \} \) and \( \eta \) be \( \alpha \mapsto (\text{nat}, \rho, R) \).

We have \( (Z, V_2) \in \mathcal{V}[\alpha]_\eta \).

We also have \( (S, V_1) \in \mathcal{V}[\alpha \rightarrow \alpha]_\eta \). (A key to the proof.)
Applications

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

**Fact** Let $\text{nat}$ be $\forall \alpha. (\alpha \to \alpha) \to \alpha \to \alpha$. If $M : \text{nat}$, then $M \cong_{\text{nat}} N_n$ 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 $\rho$ 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, $M \rho V_1 V_2 \sim_{\rho} V_1^n V_2$ (1), since $N_n \rho V_1 V_2$ reduces to $V_1^n V_2$. Let $\rho$ and $V_1 : \rho \to \rho$ and $V_2 : \rho$ be fixed.

Let $Z$ be $N_0 \text{nat}$ and $S$ be $N_1 \text{nat}$. Let $R$ in $\mathcal{R}(\text{nat}, \rho)$ be $\{(S^k Z, V_1^k V_2) \mid k \in \mathbb{N}\}$ and $\eta$ be $\alpha \mapsto (\text{nat}, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\alpha]_\eta$.

We also have $(S, V_1) \in \mathcal{V}[\alpha \to \alpha]_\eta$.

(A key to the proof.)
Applications

Inhabitants of $\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

**Fact** Let $nat$ be $\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$. If $M : nat$, then $M \simeq_{nat} N_n$ for some integer $n$, where $N_n \triangleq \Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. \lambda x : \alpha. f^n x$.

**Proof** By *extensionality*, it suffices to show that there exists $n$ such that for any closed type $\rho$ and closed values $V_1 : \rho \rightarrow \rho$ 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, $M \rho V_1 V_2 \sim_{\rho} V^n_1 V_2$ (1), since $N_n \rho V_1 V_2$ reduces to $V^n_1 V_2$. Let $\rho$ and $V_1 : \rho \rightarrow \rho$ and $V_2 : \rho$ be fixed.

Let $Z$ be $N_0 nat$ and $S$ be $N_1 nat$. Let $R$ in $R(nat, \rho)$ be $\{ (S^k Z, V_1^k V_2) \mid k \in \mathbb{N} \}$ and $\eta$ be $\alpha \mapsto (nat, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\alpha]_{\eta}$.

We also have $(S, V_1) \in \mathcal{V}[\alpha \rightarrow \alpha]_{\eta}$. (A key to the proof.)

Indeed, assume $(W_1, W_2)$ in $\mathcal{V}[\alpha]_{\eta}$. There exists $k$ such that $W_1 = S^k Z$ and $W_2 = V_1^k V_2$. Thus, $(S W_1, V_1 W_2)$ equal to $(S^{k+1} Z, V_1^{k+1} V_2)$ is in $\mathcal{E}[\alpha]_{\eta}$.
Applications

**Fact** Let \( \text{nat} \) be \( \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \). If \( M : \text{nat} \), then \( M \cong_{\text{nat}} N_n \) for some integer \( n \), where \( N_n \triangleright= \Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. \lambda x : \alpha. f^n x \).

**Proof** By *extensionality*, it suffices to show that there exists \( n \) such that for any closed type \( \rho \) and closed values \( V_1 : \rho \rightarrow \rho \) 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, \( M \rho V_1 V_2 \sim_{\rho} V_1^n V_2 (1) \), since \( N_n \rho V_1 V_2 \) reduces to \( V_1^n V_2 \). Let \( \rho \) and \( V_1 : \rho \rightarrow \rho \) and \( V_2 : \rho \) be fixed.

Let \( Z \) be \( N_0 \text{nat} \) and \( S \) be \( N_1 \text{nat} \). Let \( R \) in \( \mathcal{R}(\text{nat}, \rho) \) be \( \{(S^k Z, V_1^k V_2) \mid k \in \mathbb{N}\} \) and \( \eta \) be \( \alpha \mapsto (\text{nat}, \rho, R) \).

We have \( (Z, V_2) \in \mathcal{V}[\alpha]_\eta \).

We also have \( (S, V_1) \in \mathcal{V}[\alpha \rightarrow \alpha]_\eta \).  

(A key to the proof.)

Indeed, assume \( (W_1, W_2) \) in \( \mathcal{V}[\alpha]_\eta \). There exists \( k \) such that \( W_1 = S^k Z \) and \( W_2 = V_1^k V_2 \). Thus, \( (S W_1, V_1 W_2) \) equal to \( (S^{k+1} Z, V_1^{k+1} V_2) \) is in \( \mathcal{E}[\alpha]_\eta \).
Applications

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

**Fact** Let $nat$ be $\forall \alpha. (\alpha \to \alpha) \to \alpha \to \alpha$. If $M : nat$, 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 $\rho$ 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, $M \rho V_1 V_2 \sim_{\rho} V_1^n V_2$ (1), since $N_n \rho V_1 V_2$ reduces to $V_1^n V_2$. Let $\rho$ 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 $\mathcal{R}(nat, \rho)$ be $\{(S^k Z, V_1^k V_2) \mid k \in \mathbb{N}\}$ and $\eta$ be $\alpha \mapsto (nat, \rho, R)$.

We have $(Z, V_2) \in \mathcal{V}[\alpha]_\eta$.

We also have $(S, V_1) \in \mathcal{V}[\alpha \to \alpha]_\eta$. (A key to the proof.)

By parametricity, we have $M \sim_{nat} M$. Hence, $(M \text{ nat } S Z, M \rho V_1 V_2) \in \mathcal{E}[\alpha]_\eta$. Thus, there exists $n$ such that $M \text{ nat } S Z \cong_{nat} S^n Z$ and $M \rho V_1 V_2 \cong_{\rho} V_1^n V_2$.

Since, $M \text{ nat } S Z$ 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 $\forall \alpha. \alpha \rightarrow (\tau \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha$

▷ Left as an exercise...
Applications

\[ \forall \alpha. \alpha \rightarrow (\tau \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \]

**Fact** Let \( \tau \) be closed and \textit{list} be \( \forall \alpha. \alpha \rightarrow (\tau \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \). Let \( C \) be \( \lambda H : \tau . \lambda T : \text{list} . \Lambda \alpha . \lambda n : \alpha . \lambda c : \tau \rightarrow \alpha \rightarrow \alpha . c H (T \alpha n c) \) and \( N \) be \( \Lambda \alpha . \lambda n : \alpha . \lambda c : \tau \rightarrow \alpha \rightarrow \alpha . n \). If \( M : \text{list} \), then \( M \cong_{\text{list}} N_n \) for some \( N_n \) in \( \mathcal{L}_n \) where \( \mathcal{L}_k \) is defined inductively by

\[ \mathcal{L}_0 \triangleq \{ N \} \quad \text{and} \quad \mathcal{L}_{k+1} \triangleq \{ C W_k N_k \mid W_k \in \text{Val}(\tau) \land N_k \in \mathcal{L}_k \} \]

**Proof** By \textit{extensionality}, it suffices to show that there exists \( n \) and \( N_n \in \mathcal{L}_n \) such that for any closed type \( \rho \) and closed values \( V_1 : \tau \rightarrow \rho \rightarrow \rho \) and \( V_2 : \rho \), we have \( M \rho V_1 V_2 \sim_{\rho} N_n \rho V_1 V_2 \), or, by closure by inverse reduction and replacing observational by logical equivalence, \( C W_n (\ldots (C W_1 N) \ldots) \) (1), since \( N_n \rho V_1 V_2 \) reduces to \( C W_n (\ldots (C W_1 N) \ldots) \) where all \( W_k \) are in \( \text{Val}(\tau) \).

Let \( \rho \) and \( V_1 : \alpha \rightarrow \rho \rightarrow \rho \) and \( V_2 : \rho \) be fixed.

Let \( R \) in \( \mathcal{R}(\text{list}, \rho) \) be defined inductively as \( \bigcup R_n \) where \( R_{k+1} \) is

\[ \{ \downarrow (C G T, V_2 H U) \mid (G, H) \in \mathcal{V}[\tau]_\eta \land (T, U) \in R_k \} \]

and \( R_0 \) is \( \{(N, V_1)\} \).

We have \( (N, V_1) \in R_0 \subseteq \mathcal{V}[\alpha]_\eta \).

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

By parametricity, we have \( M \equiv \lambda \alpha. \alpha \rightarrow (\tau \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \). Hence, \( (M \text{list} C N, M \circ V_1 V_2) \in \mathcal{E}[\alpha]^{331(4)}_{412} \).\[\Box\]
Applications

\[ \forall \alpha. \alpha \to (\tau \to \alpha \to \alpha) \to \alpha \]

**Fact** Let \( \tau \) be closed and \textit{list} be \( \forall \alpha. \alpha \to (\tau \to \alpha \to \alpha) \to \alpha \). Let \( C \) be \( \lambda H : \tau. \lambda T : \text{list}. \Lambda \alpha. \lambda n : \alpha. \lambda c : \tau \to \alpha \to \alpha. c H (T \alpha n c) \) and \( N \) be \( \Lambda \alpha. \lambda n : \alpha. \lambda c : \tau \to \alpha \to \alpha. n \). If \( M : \text{list} \), then \( M \cong_{\text{list}} N_n \) for some \( N_n \) in \( \mathcal{L}_n \) where \( \mathcal{L}_k \) is defined inductively by

\[ \mathcal{L}_0 \triangleq \{ N \} \text{ and } \mathcal{L}_{k+1} \triangleq \{ C W_k \, N_k \mid W_k \in \text{Val}(\tau) \land N_k \in \mathcal{L}_k \} \]

**Proof** By \textit{extensionality}, it suffices to show that there exists \( n \) and \( N_n \in \mathcal{L}_n \) such that for any closed type \( \rho \) and closed values \( V_1 : \tau \to \rho \to \rho \) and \( V_2 : \rho \), we have \( M \rho \, V_1 \, V_2 \sim_\rho N_n \rho \, V_1 \, V_2 \), or, by closure by inverse reduction and replacing observational by logical equivalence, \( C \, W_n \, (\ldots (C \, W_1 \, N) \ldots) \) \((1)\), since \( N_n \rho \, V_1 \, V_2 \) reduces to \( C \, W_n \, (\ldots (C \, W_1 \, N) \ldots) \) where all \( W_k \) are in \( \text{Val}(\tau) \).

Let \( \rho \) and \( V_1 : \alpha \to \rho \to \rho \) and \( V_2 : \rho \) be fixed.

Let \( R \) in \( \mathcal{R}(\text{list}, \rho) \) be defined inductively as \( \bigcup R_n \) where \( R_{k+1} \) is

\[ \{ \downarrow (C \, G \, T, V_2 \, H \, U) \mid (G, H) \in \mathcal{V}[\tau]_\eta \land (T, U) \in R_k \} \text{ and } R_0 = \{ (N, V_1) \} \]

We have \((N, V_1) \in R_0 \subseteq \mathcal{V}[\alpha]_\eta \).

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

Indeed, assume \((C, H) \in \mathcal{V}[\tau]_\eta \) and \((T, U) \in \mathcal{V}[\alpha]_\eta \), i.e. in \( R_k \) for some \( k \).
Applications

\( \forall \alpha. \alpha \rightarrow (\tau \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \)

**Fact**  Let \( \tau \) be closed and \( \text{list} \) be \( \forall \alpha. \alpha \rightarrow (\tau \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \). Let \( C \) be \( \lambda H : \tau. \lambda T : \text{list}. \Lambda \alpha. \lambda n : \alpha. \lambda c : \tau \rightarrow \alpha \rightarrow \alpha. c \ H \ (T \ \alpha \ n \ c) \) and \( N \) be \( \Lambda \alpha. \lambda n : \alpha. \lambda c : \tau \rightarrow \alpha \rightarrow \alpha. n \). If \( M : \text{list} \), then \( M \equiv_{\text{list}} N_n \) for some \( N_n \) in \( \mathcal{L}_n \) where \( \mathcal{L}_k \) is defined inductively by

\[
\mathcal{L}_0 \triangleq \{N\} \quad \text{and} \quad \mathcal{L}_{k+1} \triangleq \{C \ W_k \ N_k \mid W_k \in \text{Val}(\tau) \land N_k \in \mathcal{L}_k\}
\]

**Proof** By extensionality, it suffices to show that there exists \( n \) and \( N_n \in \mathcal{L}_n \) such that for any closed type \( \rho \) and closed values \( V_1 : \tau \rightarrow \rho \rightarrow \rho \) and \( V_2 : \rho \), we have \( M \ \rho \ V_1 \ V_2 \sim_{\rho} N_n \ \rho \ V_1 \ V_2 \), or, by closure by inverse reduction and replacing observational by logical equivalence, \( C \ W_n (\ldots (C \ W_1 \ N) \ldots) \) (1), since \( N_n \ \rho \ V_1 \ V_2 \) reduces to \( C \ W_n (\ldots (C \ W_1 \ N) \ldots) \) where all \( W_k \) are in \( \text{Val}(\tau) \).

Let \( \rho \) and \( V_1 : \alpha \rightarrow \rho \rightarrow \rho \) and \( V_2 : \rho \) be fixed.

Let \( R \) in \( \mathcal{R}(\text{list}, \rho) \) be defined inductively as \( \bigcup R_n \) where \( R_{k+1} \) is \( \{\downarrow (C \ G \ T, V_2 \ H \ U) \mid (G, H) \in \mathcal{V}[\tau]_\eta \land (T, U) \in R_k\} \) and \( R_0 \) is \( \{(N, V_1)\} \).

We have \((N, V_1) \in R_0 \subseteq \mathcal{V}[\alpha]_\eta\).

We also have \((C, V_2) \in \mathcal{V}[\tau \rightarrow \alpha \rightarrow \alpha]_\eta\).  

(A key to the proof)

By parametricity, we have \( M \equiv_{\text{list}} M \). Hence, \((M \ \text{list} \ C \ N, M \ \rho \ V_1 \ V_2) \in \mathcal{E}[\eta]_{\alpha}^{\omega_{\alpha}} \)
Contents

- Introduction
- Normalization of $\lambda_{st}$
- Observational equivalence in $\lambda_{st}$
- Logical relations in stlc
- Logical relations in $F$
- Applications
- Extensions
Encodable features

Natural numbers

We have shown that all expressions of type \textit{nat} behave as natural numbers. Hence, natural numbers are definable.

Still, we could also provide a type \textit{nat} of natural numbers as primitive.

Then, we may extend

- **behavioral equivalence**: if $M_1 : \textit{nat}$ and $M_2 : \textit{nat}$, we have $M_1 \simeq_{\textit{nat}} M_2$ iff there exists $n : \textit{nat}$ such that $M_1 \Downarrow n$ and $M_2 \Downarrow n$.

- **logical equivalence**: $\mathcal{V}[\textit{nat}] \triangleq \{ (n, n) \mid n \in \mathbb{N} \}$

All properties are preserved.
Encodable features

Products

Given closed types $\tau_1$ and $\tau_2$, we defined

$$\tau_1 \times \tau_2 \triangleq \forall \alpha. (\tau_1 \to \tau_2 \to \alpha) \to \alpha$$

$$(M_1, M_2) \triangleq \Lambda \alpha. \lambda x : \tau_1 \to \tau_2 \to \alpha. x \ M_1 \ M_2$$

$$M.i \triangleq M \ (\lambda x_1 : \tau_1. \lambda x_2 : \tau_2. x_i)$$

Facts

If $M : \tau_1 \times \tau_2$, then $M \cong_{\tau_1 \times \tau_2} (M_1, M_2)$ for some $M_1 : \tau_1$ and $M_2 : \tau_2$.

If $M : \tau_1 \times \tau_2$ and $M.1 \cong_{\tau_1} M_1$ and $M.2 \cong_{\tau_2} M_2$, then $M \cong_{\tau_1 \times \tau_2} (M_1, M_2)$

Primitive pairs

We may instead extend the language with primitive pairs. Then,

$$\mathcal{V}[\tau \times \sigma]_\eta \triangleq \left\{ ((V_1, W_1), (V_2, W_2)) \right\}$$

$$\left| (V_1, V_2) \in \mathcal{V}[\tau]_\eta \land (W_1, W_2) \in \mathcal{V}[\sigma]_\eta \right\}$$
Sums

We define:

\[ \mathcal{V}[\tau + \sigma]_\eta = \{ (inj_1 V_1, inj_1 V_2) \mid (V_1, V_2) \in \mathcal{V}[\tau]_\eta \} \cup \{ (inj_2 W_1, inj_2 W_2) \mid (W_1, W_2) \in \mathcal{V}[\sigma]_\eta \} \]

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

We recursively\(^1\) define \(\mathcal{V}[\text{list } \tau]_{\eta}\) as \(\bigcup_k \mathcal{W}^k_{\eta}\) where \(\mathcal{W}^0_{\eta}\) is \(\{(\text{Nil}, \text{Nil})\}\) and \(\mathcal{W}^{k+1}_{\eta}\) is
\[
\{(\text{Cons } H_1 T_1, \text{Cons } H_2 T_2) \mid (H_1, H_2) \in \mathcal{V}[\alpha]_{\eta} \land (T_1, T_2) \in \mathcal{W}^k_{\eta}\}.
\]
Assume that \((\alpha \mapsto \rho_1, \rho_2, R) \in \eta\) where \(R\) in \(R(\rho_1, \rho_2)\) is the graph \(\langle g \rangle\) of a function \(g\), i.e. equal to \(\{(V_1, V_2) \mid g V_1 \downarrow V_2\}\). Then, we have:

\[
\mathcal{V}[\text{list } \alpha]_{\eta}(W_1, W_2) \iff \exists k, \forall \left\{\begin{array}{l}
W_1 = \text{Nil} \land W_2 = \text{Nil} \\
W_1 = \text{Cons } H_1 T_1 \land W_2 = \text{Cons } H_2 T_2 \land g H_1 \downarrow H_2 \\
\land (T_1, T_2) \in \mathcal{W}^k_{\eta}
\end{array}\right.

\iff \text{map } \rho_1 \rho_2 g \ W_1 \downarrow W_2
\]

\(^1\)This definition is well-founded.
Applications

\[ \text{sort}: \forall \alpha. (\alpha \to \alpha \to \text{bool}) \to \text{list } \alpha \]

**Fact:** Assume \( \text{sort} : \forall \alpha. (\alpha \to \alpha \to \text{bool}) \to \text{list } \alpha \to \text{list } \alpha \) (1). Then

\[
(\forall x, y, \ cmp_2 \ (f \ x) \ (f \ y) = cmp_1 \ x \ y) \implies \\
\forall \ell, \ \text{sort } \ cmp_2 \ (\text{map } f \ \ell) = \text{map } f \ (\text{sort } cmp_1 \ \ell)
\]
Applications

\( \text{sort} : \forall \alpha. (\alpha \to \alpha \to \text{bool}) \to \text{list} \alpha \)

**Proof:** Assume \( \forall x, y, \ cp (f \ x) (f \ y) \equiv cp \ x \ y \) (H).

We have \( sort \sim_{\sigma} sort \) where \( \sigma \) is \( \forall \alpha. (\alpha \to \alpha \to \text{bool}) \to \text{list} \alpha \to \text{list} \alpha \).

Thus, for all \( \rho_1, \rho_2 \), and relations \( R \) in \( R(\rho_1, \rho_2) \),

\[
\forall (cp_1, cp_2) \in V[\alpha \to \alpha \to B]_{\eta}, \quad \forall (V_1, V_2) \in V[\text{list} \alpha]_{\eta}, \quad (\text{sort } \rho_1 \ cp_1 \ V_1, \text{sort } \rho_2 \ cp_2 \ V_2) \in E[\text{list} \alpha]_{\eta}
\]

(1)

where \( \eta \) is \( \alpha \mapsto (\rho_1, \rho_2, R) \). We may choose \( R \) to be \( \langle f \rangle \) for some \( f \).

We have (1). Indeed, for all \( (V_1, V_2) \) and \( (W_1, W_2) \) in \( \langle f \rangle \), we have \( f \ V_1 \downarrow V_1 \) and \( f \ W_1 \downarrow W_1 \), hence \( cp_2 \ (f \ V_1)(f \ W_1) \downarrow cp_1 \ V_2 W_2 \). Thus \( cp_2 \ (f \ V_1)(f \ W_1) \equiv cp_1 \ V_2 W_2 \). With (H), this implies \( cp_2 \ V_1 W_1 \equiv cp_1 \ V_2 W_2 \), i.e. \( cp_2 \ V_1 W_1 \sim cp_1 \ V_2 W_2 \) since we are at type B, as expected. Hence (2) holds.

Since

\[ V[\text{list} \alpha]_{\eta} \triangleq \langle \text{map } \rho_1 \rho_2 f \rangle \subseteq V[\rho_1] \times V[\rho_2] \]

(2) reads

\[
\forall V : \text{list} \rho_1, V_2 :: \text{list} \rho_2, \quad \text{map } \rho_1 \rho_2 f V \downarrow V_2 \implies \exists W_1, W_2, \quad \left\{ \begin{array}{l}
\text{map } \rho_1 \rho_2 f W_1 \\
\text{sort } \rho_1 cp_1 V \downarrow W_1 \\
\text{sort } \rho_2 cp_2 V_2 
\end{array} \right. \approx
\]

338 412
Applications

whoami : \forall \alpha. \text{list } \alpha \rightarrow \text{list } \alpha

Left as an exercise...
Existential types

We define:

\[ \mathcal{V}[\exists \alpha. \tau]_\eta \triangleq \left\{ (\text{pack } V_1, \rho_1 \text{ as } \exists \alpha. \tau, \text{pack } V_2, \rho_2 \text{ as } \exists \alpha. \tau) \mid \right. \\
\exists \rho_1, \rho_2, R \in \mathcal{R}(\rho_1, \rho_2), \ (V_1, V_2) \in \mathcal{E}[\tau]_{\eta, \alpha \mapsto (\rho_1, \rho_2, R)} \left. \right\} \]

Compare with

\[ \mathcal{V}[\forall \alpha. \tau]_\eta = \left\{ (\Lambda \alpha. M_1, \Lambda \alpha. M_2) \mid \right. \\
\forall \rho_1, \rho_2, R \in \mathcal{R}(\rho_1, \rho_2), \ (\Lambda \alpha. M_1) \rho_1, (\Lambda \alpha. M_2) \rho_2) \in \mathcal{E}[\tau]_{\eta, \alpha \mapsto (\rho_1, \rho_2, R)} \left. \right\} \]
Consider $V_1 \triangleq (\text{not}, \text{tt})$, and $V_2 \triangleq (\text{succ}, 0)$ and $\sigma \triangleq (\alpha \to \alpha) \times \alpha$. Let $R \in \mathcal{R}(\text{bool, nat})$ be $\{(\text{tt}, 2n), (\text{ff}, 2n + 1) \mid n \in \mathbb{N}\}$ and $\eta$ be $\alpha \mapsto (\text{bool, nat, } R)$. We have $(V_1, V_2) \in \mathcal{V}[\sigma]_{\eta}$.

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

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

We have $(\text{tt, 0}) \in \mathcal{V}[\alpha]_{\eta}$, since $(\text{tt, 0}) \in R$.

We also have $(\text{not, succ}) \in \mathcal{V}[\alpha \to \alpha]_{\eta}$, which proves (1).

Indeed, assume $(W_1, W_2) \in \mathcal{V}[\alpha]_{\eta}$. Then $(W_1, W_2)$ is either of the form

- $(\text{tt, 2n})$ and $(\text{not } W_1, \text{succ } W_2)$ reduces to $(\text{ff, 2n + 1})$, or
- $(\text{ff, 2n + 1})$ and $(\text{not } W_1, \text{succ } W_2)$ reduces to $(\text{tt, 2n + 2})$.

In both cases, $(\text{not } W_1, \text{succ } W_2)$ reduces to a pair in $R$.

Hence, $(\text{not } W_1, \text{succ } W_2) \in \mathcal{E}[\alpha]_{\eta}$. 

---

**Example**

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

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

Hence, $(\text{pack } V_1, \text{bool as } \exists \alpha. \sigma, \text{ pack } V_2, \text{nat as } \exists \alpha. \sigma) \in \mathcal{V}[\exists \alpha. \sigma]$.
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 $\rho_1$ and $\rho_2$.

A client $M$ has type $\forall \alpha. \tau \rightarrow \sigma$ with $\alpha \notin \text{fv}(\sigma)$; it must use the argument parametrically, and the result is independent of the witness type.

Assume that $\rho_1$ and $\rho_2$ are two closed representation types and $R$ is in $\mathcal{R}(\rho_1, \rho_2)$. Let $\eta$ 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 $\alpha$ is not free in $\sigma$).

That is, $M \rho_1 N_1 \simeq_{\sigma} M \rho_2 N_2$: the behavior with the implementation $N_1$ with representation type $\rho_1$ is indistinguishable from the behavior with the implementation $N_2$ with representation type $\rho_2$. 
How do we deal with recursive types?

Assume that we allow equi-recursive types.

\[ \tau ::= \ldots | \mu \alpha.\tau \]

A naive definition would be

\[ \mathcal{V}[\mu \alpha.\tau]_\eta = \mathcal{V}[\alpha \mapsto \mu \alpha.\tau]_\eta \]

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 $\tau$ up to $n$ reduction steps. Omitting typing clauses:

$$
\begin{align*}
\mathcal{V}_k[B]_\eta &= \{(tt, tt), (ff, ff)\} \\
\mathcal{V}_k[\tau \rightarrow \sigma]_\eta &= \{(V_1, V_2) \mid \forall j < k, \forall (W_1, W_2) \in \mathcal{V}_j[\tau]_\eta, (V_1 W_1, V_2 W_2) \in \mathcal{E}_j[\sigma]_\eta\} \\
\mathcal{V}_k[\alpha]_\eta &= \eta_R(\alpha).k \\
\mathcal{V}_k[\forall \alpha. \tau]_\eta &= \{(V_1, V_2) \mid \forall \rho_1, \rho_2, R \in \mathcal{R}^k(\rho_1, \rho_2), \forall j < k, (V_1 \rho_1, V_2 \rho_2) \in \mathcal{V}_j[\tau]_\eta, \alpha \mapsto (\rho_1, \rho_2, R)\} \\
\mathcal{V}_k[\mu \alpha. \tau]_\eta &= \mathcal{V}_{k-1}[\alpha \mapsto \mu \alpha. \tau]_\eta \\
\mathcal{E}_k[\tau]_\eta &= \{(M_1, M_2) \mid \forall j < k, M_1 \downarrow_j V_1 \\
&\qquad \quad \implies \exists V_2, M_2 \downarrow V_2 \land (V_1, V_2) \in \mathcal{V}_{k-j}[\tau]_\eta\}
\end{align*}
$$

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 $\rho_1$ and $\rho_2$ of length (at least) $j$. 

---

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 $\rho_1$ and $\rho_2$ of length (at least) $j$. 

---

344 412
Step-indexed logical relations

(a taste)

The relation is asymmetric.

If $\Delta; \Gamma \vdash M_1, M_2 : \tau$ we define $\Delta; \Gamma \vdash M_1 \preceq M_2 : \tau$ as

$$\forall \eta \in R^k_\Delta(\delta_1, \delta_2), \forall (\gamma_1, \gamma_2) \in G_k[\Gamma], (\gamma_1(\delta_1(M_1)), \gamma_2(\delta_2(M_2)) \in E_k[\tau]_\eta$$

and

$$\Delta; \Gamma \vdash M_1 \sim M_2 : \tau \triangleq \bigwedge \begin{cases} \Delta; \Gamma \vdash M_1 \preceq M_2 : \tau \\ \Delta; \Gamma \vdash M_2 \preceq M_1 : \tau \end{cases}$$

Notations and proofs get a bit involved...

Notations may be simplified by introducing a $\textit{later}$ guard $\triangleright$ to capture incrementation of the index and avoid the explicit manipulation of integers (but the meaning remains the same).
Logical relations for $F^\omega$?

Logical relations can be generalized to work for $F^\omega$, indeed.

There is a slight complication though in the interpretation of type functions.

This is out of this course scope, but one may, for instance, read [Atkey, 2012].
Side effects, References, Value restriction
Contents

- Introduction
- Exceptions
- References in $\lambda_{st}$
- Polymorphism and references
Referential transparency

What is it?
An expression is *referentially transparent* or *pure* if it can be replaced with its corresponding value without changing the program behavior. Applying a pure function to the same arguments returns the same result.

Why is it useful?
Allows to reason about programs as a rewrite system, which may help

- prove the correction,
- perform code optimization.
- typically, it allows for: memoization, common expression elimination, lazy evaluation, . . .
- with *code parallelization, optimistic evaluation, transactions, . . .*
Referential transparency

Examples of impure constructs

- Exceptions, References, reading/printing functions.
- Interaction with the file system.
- Date and random primitives, etc.

Termination?
According to the definition, the status of termination is unclear. (As they never return, they cannot actually be replaced by the result of their evaluation—except in Haskell that uses an explicit bottom value \( \bot \).) Non-termination is usually considered impure: it breaks equational reasoning and most program transformations, as other impure constructs.

In practice, high-complexity is not so different from non-termination...

Effects
Any source of impurity is usually called an effect.
Referential transparency

**Effects are unavoidable**

Any programming language must have some impure aspects to communicate with the operating system.

Side effects may sometimes be encapsulated, *e.g.* a module with side effects may sometimes have a pure interface.

**Mitigation of effects**

So the questions are more whether:

- a large core of the language is pure/effect free (*e.g.* Haskell, Coq, Core System F) or effectful (most other languages); and/or
- side effects can be tracked, *e.g.* by the type system. (*Haskell, Koka, Rust, Mezzo, or algebraic effects*)
The semantics of effects

Programs with effects cannot be described as a pure rewrite system.

- The semantics must be changed.
- Some of the properties will be lost

We shall see:

- Exceptions, which require a small change to the semantics
- References, which:
  - require a major change to the semantics
  - do not fit well with polymorphism—which needs to be restricted in the presence of effects.
- Values, or a larger class of *non-expansive expressions*, whose evaluation is effect free play a key role in the presence of effects.

In the presence of effects, deterministic, call-by-value semantics is always a huge source of simplification when not a requirement.
Contents

- Introduction
- Exceptions
- References in $\lambda_{st}$
- Polymorphism and references
Exceptions

Exceptions are a mechanism for changing the normal order of evaluation usually, but not necessarily, in case something abnormal occurred.

When an exception is raised, the evaluation does not continue as usual: Shortcutting normal evaluation rules, the exception is propagated up into the evaluation context until some handler is found at which the evaluation resumes with the exceptional value received; if no handler is found, the exception had reached the toplevel and the result of the evaluation is the exception instead of a value.

We extend the language with a constructor form to raise an exception and a destructor form to catch an exception; we also extend the evaluation contexts:

\[ M ::= \ldots | \text{raise } M | \text{try } M \text{ with } M \]
\[ E ::= \ldots | \text{raise } [] | \text{try } [] \text{ with } M \]
Exceptions

We do not treat $\text{raise } V$ as a value, since it stops the normal order of evaluation. Instead, reduction rules propagate and handle exceptions:

$\text{Raise}$

$$F[\text{raise } V] \rightarrow \text{raise } V$$

$\text{Handle-Val}$

$$\text{try } V \text{ with } M \rightarrow V$$

$\text{Handle-Raise}$

$$\text{try } \text{raise } V \text{ with } M \rightarrow M \ V$$

Rule $\text{Raise}$ uses an evaluation context $F$ which stands for any $E$ other than $\text{try } [] \text{ with } M$, so that it propagates an exception up the evaluation contexts, but not through a handler.

The case of the handler is treated by two specific rules:

- Rule $\text{Handle-Raise}$ passes an exceptional value to its handler;
- Rule $\text{Handle-Val}$ removes the handler around a value.
Exceptions

For example, assuming that $K$ is $\lambda x.\lambda y. y$ and $M \rightarrow V$, we have the following reduction:

$$
\begin{align*}
\text{try } K \ (\text{raise } M) \ & \text{ with } \lambda x. x & \quad \text{by CONTEXT} \\
\rightarrow \ & \text{try } K \ (\text{raise } V) \ & \text{ with } \lambda x. x & \quad \text{by RAISE} \\
\rightarrow \ & \text{try } \text{raise } V \ & \text{ with } \lambda x. x & \quad \text{by HANDLE-RAISE} \\
\rightarrow \ & (\lambda x. x) \ V \\
\rightarrow \ & V 
\end{align*}
$$

In particular, we do not have the following step,

$$
\begin{align*}
\text{try } K \ (\text{raise } V) \ & \text{ with } \lambda x. x & \quad \text{by } \beta_v \\
\rightarrow & \text{try } \lambda y. y \ & \text{ with } \lambda x. x \rightarrow \lambda y. y
\end{align*}
$$

since $\text{raise } V$ is not a value, so the first $\beta$-reduction step is not allowed.
Exceptions

We assume given a fixed type $\tau_{\text{exn}}$ for exceptional values.

\[
\text{Raise} \quad \frac{\Gamma \vdash M : \tau_{\text{exn}}}{\Gamma \vdash \text{raise } M : \tau}
\]

\[
\text{Try} \quad \frac{\Gamma \vdash M_1 : \tau \quad \Gamma \vdash M_2 : \tau_{\text{exn}} \to \tau}{\Gamma \vdash \text{try } M_1 \text{ with } M_2 : \tau}
\]

There are some subtleties:

- Raise turns an expression of type $\tau_{\text{exn}}$ into an exception.
- Consistently, the handler has type $\tau_{\text{exn}} \to \tau$, since it receives the exception value of type $\text{exn}$ as argument;
- An exceptional value of type $\text{exn}$ may be raised in $M_1$ and used in $M_2$ without any visible flow at the type level. Hence, $\text{raise} \cdot$ and $\text{try} \cdot \text{with} \cdot$ must agree on the type $\text{exn}$.
- Both premises of Rule $\text{TRY}$ must return values of the same type $\tau$.
- $\text{raise } M$ can have any type, as the current computation is aborted.
Exceptions

What should we choose for \( \tau_{exn} \)? Well, any type:

- Choosing `unit`, exceptions will not carry any information.
- Choosing `int`, exceptions can report some error code.
- Choosing `string`, exceptions can report error messages.
- Using a sum type or better a variant type (tagged sum), with one case to describe each exceptional situation.

This is the approach followed by ML, which declares a new extensible type `exn` for exceptions: this is a sum type, except that all cases are not declared in advance, but only as needed. (Extensible datatypes are available in OCaml since version 4.02.)

In all cases, the type of exception must be fixed in the whole program.

This is because `raise` \( \cdot \) and `try` \( \cdot \) `with` \( \cdot \) must agree beforehand on the type of exceptions as this type is not passed around by the typing rules.
Encoding of multiple exceptions

Introduce a data type:

\[
\text{type } \text{exn} = \sum (E_i : \tau_i \to \text{exn})_{i \in I}
\]

Use syntactic sugar:

\[
\begin{align*}
\text{raise } E_i \ v & \triangleq \text{raise} (E_i \ v) \\
\text{try } M \ \text{with} \ (E_j \ x \ \Rightarrow \ M_k)_{j \in J} & \triangleq \text{try} \ M \ \text{with} \\
& \quad (\lambda z. \ \text{match} \ z \ \text{with} \ (E_j \ x \ \Rightarrow \ M_k)_{j \in J} | \ z \ \Rightarrow \ \text{raise} \ z)
\end{align*}
\]
Exceptions

How do we state type soundness, since exceptions may be uncaught?

By saying that this is the only “exception” to progress:

**Theorem (Progress)**

A well-typed, irreducible term is either a value or an uncaught exception. If $\emptyset \vdash M : \tau$ and $M \rightarrow$, then $M$ is either $V$ or raise $V$ for some value $V$. 
Exceptions

Structured exceptions

What is the type \textit{exn} for exceptions? Well, it could be any type:

- If we take the unit type, we only know that an except has been raised but cannot pass any other information.
- Hence, we could take the type of integers (e.g. passing error codes, much as commands do in Unix)
- Use some richer data type with one constructor per kind of error.
- Use a variant type (tagged sum)

The handler may analyze the argument of the exception.
Exceptions

An uncaught exception is often a programming error. It may be surprising that they are not detected by the type system.

Exceptions may be detected using more expressive type systems. Unfortunately, the existing solutions are often complicated for some limited benefit, and are still not often used in practice.

The complication comes from the treatment of functions, which have some latent effect of possibly raising or catching an exception when applied. To be precise, the analysis must therefore enrich types of functions with latent effects, which is quite invasive and obfuscating.

Uncaught exceptions must be declared in the language Java. (Java also has untraced exceptions.)

See Leroy and Pessaux [2000] for a solution in ML.
Exceptions

Once raised, exceptions are propagated step-by-step by Rule \texttt{Raise} until they reach a handler or the toplevel.

We can also describe their semantics by replacing propagation of exceptions by deep handling of exceptions inside terms.

Replace the three previous reduction rules by:

\begin{align*}
\texttt{Handle-Val'}: & \quad \text{try } V \text{ with } M \rightarrow V \\
\texttt{Handle-Raise'}: & \quad \text{try } F[\text{raise } V] \text{ with } M \rightarrow M \; V
\end{align*}

where $F$ is a sequence of $F$ contexts, \textit{i.e.} handler-free evaluation context of arbitrary depth.

This semantics is perhaps more intuitive, closer to what a compiler does, but the two presentations are equivalent.

In this case, uncaught exceptions are of the form $F[\text{raise } V]$. 
Benton and Kennedy [2001] have argued for merging let and try constructs into a unique form \( \text{let } x = M_1 \text{ with } M_2 \text{ in } M_3 \). The expression \( M_1 \) is evaluated first and

- if it returns a value it is substituted for \( x \) in \( M_3 \), as if we had evaluated \( \text{let } x = M_1 \text{ in } M_3 \);
- otherwise, \( i.e. \), if it raises an exception \( \text{raise } V \), then the exception is handled by \( M_2 \), as if we had evaluated \( \text{try } M_1 \text{ with } M_2 \).

This combined form captures a common programming pattern:

\[
\text{let rec } \text{read_config_in_path filename (dir :: dirs) →}
\]

\[
\text{let } \text{fd} = \text{open_in (Filename.concat dir filename)}
\]

\[
\text{with } \text{Sys_error } \_ \to \text{read_config_filename dirs in}
\]

\[
\text{read_config_from_fd } \text{fd}
\]

Workarounds are inelegant and inefficient. This form is also better suited for program transformations (see Benton and Kennedy [2001]).
Exceptions

Interesting syntactic variation

Encoding the new form \( \text{let } x = M_1 \text{ with } M_2 \text{ in } M_3 \) with “let” and “try” is not easy:

In particular, it is not equivalent to: \( \text{try let } x = M_1 \text{ in } M_3 \text{ with } M_2 \).

The continuation \( M_3 \) could raise an exception that would then be handled by \( M_2 \), which is not intended.

There are several encodings:

- Use a sum type to know whether \( M_1 \) raised an exception:
  \[
  \text{case} \ (\text{try } \text{Val} \ M_1 \text{ with } \lambda y. \text{Exc} \ y) \ of \ (\text{Val} : \lambda x. M_3 \ | \ \text{Exc} : M_2)
  \]

- Freeze the continuation \( M_3 \) while handling the exception:
  \[
  (\text{try let } x = M_1 \text{ in } \lambda(). M_3 \text{ with } \lambda y. \lambda(). M_2 \ y) ()
  \]

Unfortunately, they are both hardly readable—and inefficient.
A similar construct has been added in OCaml version 4.02, allowing exceptions combined with pattern matching.

The previous example can now be written in OCaml as:

```
let rec read_config_in_path filename path =
  match path with [] → [] | dir :: dirs →
    match open_in (Filename.concat dir filename) with
    | fd → read_config_from_fd fd
    | exception Sys_error _ → read_config_in_path filename dirs
```
Exceptions

Do all well-typed programs terminate in the presence of exceptions?

No, because exceptions hide the type of values that they communicate to the handler, which can be used to emulate recursive types.

Encode values of type $\tau_0$ as lazy values of type $\text{unit} \rightarrow \tau_0$, say $\tau$

Let $\text{encode}$ be $\text{fun} \ x \ () \rightarrow x$ and $\text{decode}$ be $\text{fun} \ x \rightarrow x ()$.

Let $\text{dummy}$ be some value of type $\tau_0$.

Let type $\text{exn}$ be $\tau \rightarrow \tau$, say $\sigma$.

Define the two coercion functions between types $\sigma$ and $\tau$:

$\text{fold} : \sigma \rightarrow \tau \triangleq \lambda f : \sigma. \lambda () . \text{let } _ = \text{raise } f \text{ in } \text{dummy}$

$\text{unfold} : \tau \rightarrow \sigma \triangleq \lambda f : \tau. \text{try let } _ = f () \text{ in } \lambda x : \tau. x \text{ with } \lambda y : \tau \rightarrow \tau. y$

We may then define $\omega \triangleq \lambda x. (\text{unfold } x) \ x$ so that $\omega (\text{fold } \omega)$ loops.

Or a call-by-value fixpoint of type $(\sigma \rightarrow \sigma) \rightarrow \sigma$ that allows recursive definition of functions of type $\tau \rightarrow \tau$ (encoding type $\tau_0 \rightarrow \tau_0$).
Exercise

Program factorial with the previous encoding without using recursion (nor recursive types, nor references)
Exercise

Semantics of $let \cdot = \cdot with \cdot in$

Describes the dynamic semantics of the $let x = M_1 with M_2 in M_3$.

Solution

We need a new evaluation context:

$$E ::= \ldots \mid let x = E with M_2 in M_3$$

and the following reduction rules:

**Raise**

$$F[\text{raise } V] \longrightarrow \text{raise } V$$

**Handle-Val**

$$let x = V with M_2 in M_3 \longrightarrow [x \mapsto V]M_3$$

**Handle-Raise**

$$let x = \text{raise } V with M_2 in M_3 \longrightarrow M_2 V$$
Exercise

A finalizer is some code that should always be run, whether the evaluation ends normally or an exception is being raised.

Write a function `try_finalize` that takes four arguments `f`, `x`, `g`, and `y` and returns the application `f x` with finalizing code `g y`. i.e. `g y` should be called before returning the result of the application of `f` to `x` whether it executed normally or raised an exception.

(You may try first without using binding mixed with exceptions, then using it, and compare.)
Exercise

(Solution to) Try finalize

Without `let · = · with · in`:

```ocaml
let finalize f x g y = 
  let result = try f x with exn → g y; raise exn in g y; result
```

An alternative version that does not duplicate the finalizing code and could be inlined, but allocates an intermediate result, is:

```ocaml```

```ocaml```

More concisely:

```ocaml```
Generalizing exceptions

Exceptions allow to abort the current computation to the dynamically enclosing handler.

Effect handlers are a variant of control operators.

As exceptions, they allow to abort the current computation to the dynamically enclosing handler, but offer the handler the possibility to resume the computation where it was aborted.

They are (much) more expressive.

They also allow to model a global state, where a toplevel heap handler is setup so that allocation, read, and write can be implemented by passing control to the handler together with the current continuation, i.e. evaluation context, which may change the heap and then resume or throw away the continuation.
Contents

- Introduction
- Exceptions
- References in $\lambda_{st}$
- Polymorphism and references
References

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

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

Expressions and evaluation contexts are extended:

\[
\begin{align*}
    M & ::= \ldots \mid \text{ref } M \mid M := M \mid ! M \\
    E & ::= \ldots \mid \text{ref } [] \mid [] := M \mid V := [] \mid ! []
\end{align*}
\]
A reference allocation is not a value. Otherwise, by $\beta$, the program:

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

(which intuitively should yield $?$) would reduce to:

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

(which yields 3).

How shall we solve this problem?
References

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

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

We extend the simply-typed $\lambda$-calculus calculus with *memory locations*:

$$V ::= \ldots | \ell$$

$$M ::= \ldots | \ell$$

A memory location is just an atom (that is, a name). The value found at a location $\ell$ is obtained by indirection through a *memory* (or *store*).

A memory $\mu$ is a finite mapping of locations to *closed* values.
A **configuration** is a pair $M / \mu$ of a term and a store. The operational semantics (given next) reduces configurations instead of expressions.

The semantics maintains a **no-dangling-pointers** invariant: the locations that appear in $M$ or in the image of $\mu$ are in the domain of $\mu$.

- Initially, the store is empty, and the term contains no locations, because, by convention, memory locations cannot appear in source programs. So, the invariant holds.
- If we wish to start reduction with a non-empty store, we must check that the initial configuration satisfies the **no-dangling-pointers** invariant.
References

Because the semantics now reduces configurations, all existing reduction rules are augmented with a store, which they do not touch:

\[(\lambda x : \tau. M) \ V / \mu \rightarrow [x \mapsto V]M / \mu\]

\[E[M] / \mu \rightarrow E[M'] / \mu' \quad \text{if } M / \mu \rightarrow M' / \mu'\]

Three new reduction rules are added:

\[\text{ref } V / \mu \rightarrow \ell / \mu[\ell \mapsto V] \quad \text{if } \ell \notin \text{dom}(\mu)\]

\[\ell := V / \mu \rightarrow () / \mu[\ell \mapsto V]\]

\[! \ell / \mu \rightarrow \mu(\ell) / \mu\]

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

The type system is modified as follows. Types are extended:

\[ \tau ::= \ldots \mid \text{ref } \tau \]

Three new typing rules are introduced:

\[ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \text{ref } M : \text{ref } \tau} \]
\[ \frac{\Gamma \vdash M_1 : \text{ref } \tau \quad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 := M_2 : \text{unit}} \]
\[ \frac{\Gamma \vdash M : \text{ref } \tau}{\Gamma \vdash ! M : \tau} \]

Is that all we need?
The preceding setup is enough to typecheck source terms, but does not allow stating or proving type soundness.

Indeed, we have not yet answered these questions:

- What is the type of a memory location $\ell$?
- When is a configuration $\mathcal{M} / \mu$ well-typed?
When does a location \( \ell \) have type \( \text{ref}\ \tau \)?

A possible answer is, *when it points to some value of type \( \tau \).*

Intuitively, this could be formalized by a typing rule of the form:

\[
\begin{align*}
\mu, \emptyset & \vdash \mu(\ell) : \tau \\
\mu, \Gamma & \vdash \ell : \text{ref}\ \tau
\end{align*}
\]

Comments?

- Typing judgments would have the form \( \mu, \Gamma \vdash M : \tau \). However, they would no longer be *inductively* defined (or else, every cyclic structure would be ill-typed). Instead, *co-induction* would be required.

- Moreover, if the value \( \mu(\ell) \) happens to admit two distinct types \( \tau_1 \) and \( \tau_2 \), then \( \ell \) admits types \( \text{ref}\ \tau_1 \) and \( \text{ref}\ \tau_2 \). So, one can write at type \( \tau_1 \) and read at type \( \tau_2 \): this rule is *unsound!*
A simpler and sound approach is to fix the type of a memory location when it is first allocated. To do so, we use a store typing $\Sigma$, a finite mapping of locations to types.

So, when does a location $\ell$ have type $\text{ref } \tau$? “When $\Sigma$ says so.”

$$\Sigma, \Gamma \vdash \ell : \text{ref } \Sigma(\ell)$$

Comments:

- Typing judgments now have the form $\Sigma, \Gamma \vdash M : \tau$. 

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

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

\[
\begin{align*}
\text{STORE} & \quad \forall \ell \in \text{dom}(\mu), \quad \Sigma, \emptyset \vdash \mu(\ell) : \Sigma(\ell) \\
\quad \vdash \mu : \Sigma
\end{align*}
\]

\[
\begin{align*}
\text{CONFIG} & \quad \vdash \mu : \Sigma \\
\quad & \quad \Sigma, \emptyset \vdash M : \tau \\
\quad & \quad \vdash M / \mu : \tau
\end{align*}
\]

Comments:

- This is an *inductive* definition. The store typing \(\Sigma\) serves both as an assumption (Loc) and a goal (Store). Cyclic stores are not a problem.
- The store typing is used only in the definition of a “well-typed configuration” and in the typechecking of locations. Thus, it is not needed for type-checking source programs, since the store is empty and the empty-store configuration is always well-typed.
- Notice that \(\Sigma\) does not appear in the conclusion of \text{CONFIG}. 

Restating type soundness

The type soundness statements are slightly modified in the presence of the store, since we now reduce configurations:

**Theorem (Subject reduction)**

*Reduction preserves types: if $M_\mu \rightarrow M'_\mu$ and $\vdash M_\mu : \tau$, then $\vdash M'_\mu : \tau$.***

**Theorem (Progress)**

*If $M_\mu$ is a well-typed, irreducible configuration, then $M$ is a value.*
Restating subject reduction

Inlining $\text{Config}$, subject reduction can also be restated as:

**Theorem (Subject reduction, expanded)**

If $M / \mu \rightarrow M' / \mu'$ and $\vdash \mu : \Sigma$ and $\Sigma, \emptyset \vdash M : \tau$, then there exists $\Sigma'$ such that $\vdash \mu' : \Sigma'$ and $\Sigma', \emptyset \vdash M' : \tau$.

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

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

The hypotheses are:

\[ M / \mu \rightarrow M' / \mu' \quad \text{and} \quad \vdash \mu : \Sigma \quad \text{and} \quad \Sigma, \emptyset \vdash E[M] : \tau \]

Assuming compositionality, there exists \( \tau' \) such that:

\[ \Sigma, \emptyset \vdash M : \tau' \quad \text{and} \quad \forall M', \quad (\Sigma, \emptyset \vdash M' : \tau') \Rightarrow (\Sigma, \emptyset \vdash E[M'] : \tau) \]

Then, by the induction hypothesis, there exists \( \Sigma' \) such that:

\[ \vdash \mu' : \Sigma' \quad \text{and} \quad \Sigma', \emptyset \vdash M' : \tau' \]

Here, we are stuck. The context \( E \) is well-typed under \( \Sigma \), but the term \( M' \) is well-typed under \( \Sigma' \), so we cannot combine them.

**How can we fix this?**
Establishing subject reduction

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

This is formalized by strengthening the subject reduction statement:

**Theorem (Subject reduction, strengthened)**

If $M / \mu \rightarrow M' / \mu'$ and $\vdash \mu : \Sigma$ and $\emptyset \vdash M : \tau$, then there exists $\Sigma'$ such that $\vdash \mu' : \Sigma'$ and $\emptyset \vdash M' : \tau$ and $\Sigma \subseteq \Sigma'$.

At each reduction step, the new store typing $\Sigma'$ extends the previous store typing $\Sigma$. 
Establishing subject reduction

Growing the store typing preserves well-typedness:

**Lemma (Stability under memory allocation)**

*If* $\Sigma \subseteq \Sigma'$ *and* $\Sigma, \Gamma \vdash M : \tau$, *then* $\Sigma', \Gamma \vdash M : \tau$.

(This is a generalization of the weakening lemma.)
Establishing subject reduction

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

**Lemma (Compositionality)**

Assume $\Sigma, \emptyset \vdash E[M] : \tau$. Then, there exists $\tau'$ such that:

- $\Sigma, \emptyset \vdash M : \tau'$,
- For every $\Sigma'$ such that $\Sigma \subseteq \Sigma'$, for every $M'$, $\Sigma', \emptyset \vdash M' : \tau'$ implies $\Sigma', \emptyset \vdash E[M'] : \tau$. 
Establishing subject reduction

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

The hypotheses are:

\[ \vdash \mu : \Sigma \quad \text{and} \quad \Sigma, \emptyset \vdash E[M] : \tau \quad \text{and} \quad M / \mu \rightarrow M' / \mu' \]

By compositionality, there exists \( \tau' \) such that:

\[ \Sigma, \emptyset \vdash M : \tau' \]

\[ \forall \Sigma', \forall M', \quad (\Sigma \subseteq \Sigma') \Rightarrow (\Sigma', \emptyset \vdash M' : \tau') \Rightarrow (\Sigma', \emptyset \vdash E[M'] : \tau') \]

By the induction hypothesis, there exists \( \Sigma' \) such that:

\[ \vdash \mu' : \Sigma' \quad \text{and} \quad \Sigma', \emptyset \vdash M' : \tau' \quad \text{and} \quad \Sigma \subseteq \Sigma' \]

The goal immediately follows.
Exercise (Recommended)

Prove subject reduction and progress for simply-typed \(\lambda\)-calculus equipped with unit, pairs, sums, recursive functions, exceptions, and references!
Monads

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

\[
\begin{align*}
\text{return} & : \alpha \rightarrow IO\alpha \\
\text{bind} & : IO\alpha \rightarrow (\alpha \rightarrow IO\beta) \rightarrow IO\beta \\
\text{main} & : IO() \\
\text{newIORef} & : \alpha \rightarrow IO(IORef\alpha) \\
\text{readIORef} & : IORef\alpha \rightarrow IO\alpha \\
\text{writeIORef} & : IORef\alpha \rightarrow \alpha \rightarrow IO() \\
\end{align*}
\]

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

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

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

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

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

See also the Mezzo language [Pottier and Protzenko, 2013] designed especially for the explicit control of resources.

A similar approach is taken in the language Rust.
Contents

- Introduction
- Exceptions
- References in $\lambda_{st}$
- Polymorphism and references
Combining extensions

We have shown how to extend simply-typed \( \lambda \)-calculus, independently, with:

- polymorphism, and
- references.

Can these two extensions be combined?
Beware of polymorphic locations!

When adding references, we noted that type soundness relies on the fact that *every reference cell (or memory location) has a fixed type*.

Otherwise, if a location had two types \( \text{ref } \tau_1 \) and \( \text{ref } \tau_2 \), one could store a value of type \( \tau_1 \) and read back a value of type \( \tau_2 \).

Hence, it should also be *unsound if a location could have type \( \forall \alpha. \text{ref } \tau \) (where \( \alpha \) appears in \( \tau \)) as it could then be specialized to both types \( \text{ref} ([\alpha \mapsto \tau_1]\tau) \) and \( \text{ref} ([\alpha \mapsto \tau_2]\tau) \).

By contrast, a location \( \ell \) can have type \( \text{ref} (\forall \alpha. \tau) \): this says that \( \ell \) stores values of polymorphic type \( \forall \alpha. \tau \), but \( \ell \), as a value, is viewed with the monomorphic type \( \text{ref} (\forall \alpha. \tau) \).
A counter example

Still, if naively extended with references, System F allows construction of polymorphic references, which breaks subject reduction:

\[
\text{let } y : \forall \alpha. \text{ref} (\alpha \to \alpha) = \Lambda \alpha. \text{ref} (\alpha \to \alpha) (\lambda z:\alpha. z) \text{ in }
\]
\[
(y \text{ bool}) := (\text{bool} \to \text{bool}) \text{ not; }
\]
\[
!(\text{int} \to \text{int}) (y \text{ int}) 1 / \emptyset
\]

\[
\* \rightarrow \text{ not } 1 / \ell \mapsto \text{ not}
\]

What happens is that the evaluation of the reference:

- creates and returns a location \(\ell\) bound to the identity function \(\lambda z:\alpha. z\) of type \(\alpha \to \alpha\),
- abstracts \(\alpha\) in the result and binds it to \(y\) with the polymorphic type \(\forall \alpha. \text{ref} (\alpha \to \alpha)\);
- writes the location at type \(\text{ref} (\text{bool} \to \text{bool})\) and reads it back at type \(\text{ref} (\text{int} \to \text{int})\).
Nailing the bug

In the counter-example, the first reduction step uses the following rule (where \( V \) is \( \lambda x:\alpha. x \) and \( \tau \) is \( \alpha \rightarrow \alpha \)).

\[
\begin{align*}
\text{CONTEXT} & \quad \text{ref} \; \tau \; V / \emptyset \quad \rightarrow \quad l / l \mapsto V \\
\Lambda \alpha. \text{ref} \; \tau \; V / \emptyset & \quad \rightarrow \quad \Lambda \alpha. l / l \mapsto V
\end{align*}
\]

While we have

\[
\alpha \vdash \text{ref} \; \tau \; V / \emptyset : \text{ref} \; \tau \quad \text{and} \quad \alpha \vdash l / l \mapsto V : \text{ref} \; \tau
\]

We have

\[
\alpha \vdash \Lambda \alpha. \text{ref} \; \tau \; V / \emptyset : \forall \alpha. \text{ref} \; \tau \quad \text{but not} \quad \alpha \vdash \Lambda \alpha. l / l \mapsto V : \forall \alpha. \text{ref} \; \tau
\]

Hence, the context case of subject reduction breaks.
Nailing the bug

The typing derivation of $\Lambda \alpha. \ell$ requires a store typing $\Sigma$ of the form $\ell : \tau$ and a derivation of the form:

$$
\Gamma \vdash \ell : \text{ref } \tau \\
\Gamma \vdash \Lambda \alpha. \ell : \forall \alpha. \text{ref } \tau
$$

However, the typing context $\Sigma, \alpha$ is ill-formed as $\alpha$ appears free in $\Sigma$.

Instead, a well-formed premise should bind $\alpha$ earlier as in $\alpha, \Sigma \vdash \ell : \text{ref } \tau$, but then, Rule $\text{TABS}$ cannot be applied.

By contrast, the expression $\text{ref } \tau \ V$ is pure, so $\Sigma$ may be empty:

$$
\alpha \vdash \text{ref } \tau \ V : \text{ref } \tau \\
\emptyset \vdash \Lambda \alpha. \text{ref } \tau \ V : \forall \alpha. \text{ref } \tau
$$

The expression $\Lambda \alpha. \ell$ is correctly rejected as ill-typed, so $\Lambda \alpha. (\text{ref } \tau \ V)$ should also be rejected.
Fixing the bug

Mysterious slogan:

*One must not abstract over a type variable that might, after evaluation of the term, enter the store typing.*

Indeed, this is what happens in our example. The type variable $\alpha$ which appears in the type $\alpha \to \alpha$ of $V$ is abstracted in front of $\text{ref} (\alpha \to \alpha) V$.

When $\text{ref} (\alpha \to \alpha) V$ reduces, $\alpha \to \alpha$ becomes the type of the fresh location $\ell$, which appears in the new store typing.

This is all well and good, but *how* do we enforce this slogan?
Fixing the bug

In the context of ML, a number of rather complex historic approaches have been followed: see Leroy [1992] for a survey.

Then came Wright [1995], who suggested an amazingly simple solution, known as the value restriction: only value forms can be abstracted over.

\[
\text{TAbs} \quad \frac{\Gamma, \alpha \vdash u : \tau}{\Gamma \vdash \Lambda \alpha. u : \forall \alpha. \tau}
\]

\[
\text{Value forms:} \quad u ::= x | V | \Lambda \alpha. u | u \tau
\]

The problematic proof case vanishes, as we now never $\beta\delta$-reduce under type abstraction—only $\iota$-reduction is allowed.

Subject reduction holds again.
A good intuition: internalizing configurations

A configuration \( M / \mu \) is an expression \( M \) in a memory \( \mu \). The memory can be viewed as a recursive extensible record.

The configuration \( M / \mu \) may be viewed as the recursive definition (of values) \( \text{let rec } m : \Sigma = \mu \text{ in } [\ell \mapsto m.\ell]M \) where \( \Sigma \) is a store typing for \( \mu \).

The store typing rules are coherent with this view.

Allocation of a reference is a reduction of the form

\[
\text{let rec } m : \Sigma = \mu \text{ in } E[\text{ref } \tau \ V] \quad \longrightarrow \quad \text{let rec } m : \Sigma, \ell : \tau = \mu, \ell \mapsto V \text{ in } E[m.\ell]
\]

For this transformation to preserve well-typedness, it is clear that the evaluation context \( E \) must not bind any free type variable of \( \tau \).

Otherwise, we are violating the scoping rules.
Clarifying the typing rules

Let us review the typing rules for configurations:

\[
\text{CONFIG} \quad \bar{\alpha}, \Sigma, \emptyset \vdash M : \tau \quad \bar{\alpha} \vdash \mu : \Sigma
\]

\[
\quad \bar{\alpha} \vdash M \backslash \mu : \tau
\]

\[
\text{STORE} \quad \forall \ell \in \text{dom}(\mu), \quad \bar{\alpha}, \Sigma, \emptyset \vdash \mu(\ell) : \Sigma(\ell)
\]

\[
\quad \bar{\alpha} \vdash \mu : \Sigma
\]

Remarks:

- Closed configurations are typed in an environment just composed of type variables \( \bar{\alpha} \).
- \( \bar{\alpha} \) may appear in the store during reduction.
  Take for example, \( M \) equal to \( \text{ref}(\alpha \rightarrow \alpha) \ V \) where \( V \) is \( \lambda x : \alpha. x \).
- Thus \( \bar{\alpha} \) will also appear in the store typing and should be placed in front of the store typing; no \( \beta \) in \( \bar{\alpha} \) can be generalized.
- New type variables cannot be introduced during reduction.
Clarifying the typing rules

Judgments are now of the form $\bar{\alpha}, \Sigma, \Gamma \vdash M : \tau$ although we may see $\bar{\alpha}, \Sigma, \Gamma$ as a whole typing context $\Gamma'$. For locations, we need a new context formation rule:

$$
\text{WFEnvLoc} \quad \Gamma, \ell : \tau
\quad \ell \notin \text{dom}(\Gamma)
\quad \Gamma \vdash \tau
\quad \Gamma \vdash \ell
\quad \Gamma, \ell \vdash \Gamma
$$

This allows locations to appear anywhere. However, in a derivation of a closed term, the typing context will always be of the form $\bar{\alpha}, \Sigma, \Gamma$ where:

- $\Sigma$ only binds locations (to arbitrary types) and
- $\Gamma$ does not bind locations.
Clarifying the typing rules

The typing rule for memory locations (where $\Gamma$ is of the form $\vec{\alpha}, \Sigma, \Gamma'$)

\[
\text{Loc} \\
\Gamma \vdash \ell : \text{ref} \Gamma(\ell)
\]

In System F, typing rules for references need not be primitive. We may instead treat them as constants of the following types:

\[
\text{ref} : \forall \alpha. \alpha \to \text{ref} \alpha \\
(!) : \forall \alpha. \text{ref} \alpha \to \alpha \\
(\:=) : \forall \alpha. \text{ref} \alpha \to \alpha \to \text{unit}
\]

There are all destructors (event \text{ref}) with the obvious arities.

The $\delta$-rules are adapted to carry explicit type parameters:

\[
\text{ref } \tau V / \mu \quad \longrightarrow \quad \ell / \mu[\ell \mapsto V] \quad \quad \text{if } \ell \notin \text{dom}(\mu) \\
\ell := (\tau) V / \mu \quad \longrightarrow \quad () / \mu[\ell \mapsto V] \\
!\tau \ell / \mu \quad \longrightarrow \quad \mu(\ell) / \mu
\]
Stating type soundness

Lemma (Subject reduction for constants)
\( \delta \)-rules preserve well-typedness of closed configurations.

Theorem (Subject reduction)
Reduction of closed configurations preserves well-typedness.

Lemma (Progress for constants)
A well-typed closed configuration \( M/\mu \) where \( M \) is a full application of constants ref, (!), or (=) to types and values can always be reduced.

Theorem (Progress)
A well-typed irreducible closed configuration \( M/\mu \) is a value.
Consequences

The problematic program is now syntactically ill-formed:

\[
\text{let } y : \forall \alpha. \text{ref } (\alpha \to \alpha) = \Lambda \alpha. \text{ref } (\lambda z : \alpha. z) \text{ in} \\
(\text{:=}) (\text{bool} \to \text{bool}) (y \text{ bool}) \text{ not};; \\
! (\text{int} \to \text{int}) (y (\text{int})) 1
\]

Indeed, \(\text{ref } (\lambda z : \alpha. z)\) is not a value form, but the application of a unary destructor to a value, so it cannot be generalized.
Value restriction

With the value restriction, some pure programs become ill-typed, even though they were well-typed in the absence of references.

Therefore, this style of introducing references in System F (or in ML) is *not a conservative extension*.

Assuming:

\[
\text{map} : \forall \alpha. \forall \beta. (\alpha \to \beta) \to \text{list} \, \alpha \to \text{list} \, \beta \quad \text{id} : \forall \alpha. \alpha \to \alpha
\]

This expression becomes ill-typed:

\[
\Lambda \alpha. \text{map} \, \alpha \, \alpha \, (\text{id} \, \alpha)
\]

A common work-around is to perform a manual \(\eta\)-expansion:

\[
\Lambda \alpha. \lambda y: \text{list} \, \alpha. \text{map} \, \alpha \, (\text{id} \, \alpha) \, y
\]

Of course, in the presence of side effects, \(\eta\)-expansion is *not* semantics-preserving, so this must not be done blindly.
Value restriction

Non-expansive expressions

The value restriction can be slightly relaxed by enlarging the class of value-forms to a syntactic category of so-called *non-expansive terms*—terms whose evaluation will definitely not allocate new reference cells. Non-expansive terms form a strict superset of value-forms.

\[
\begin{array}{c}
u \\ ::= \\ x \mid V \mid \Lambda \alpha. u \mid u \tau \\
\mid let \ x = u \ in \ u \mid (\lambda x: \tau. u) \ u \\
\mid C \ u_1 \ldots u_k \\
\mid d \ u_1 \ldots u_k \quad \text{where either} \quad \begin{cases} k < \text{arity}(d) \\ d \text{ is non-expansive.} \end{cases}
\end{array}
\]

In particular, pattern matching is a non-expansive destructor! But \( \text{ref} \cdot \) is an expansive one!.

For example, the following expression is non-exapnsive:

\[
\Lambda \alpha. \ let \ x = (\text{match} \ y \ \text{with} \ (C_i \ x_i \rightarrow u_i)_{i \in I}) \ in \ u
\]
Value restriction

Positive occurrences: Garrigue [2004] relaxes the value restriction in a more subtle way, which is justified by a subtyping argument.

For instance, let \( x : \forall \alpha. \text{list } \alpha = \Lambda \alpha. (M_1 \ M_2) \) in \( M \) may be well-typed because \( \alpha \) appears only positively in the type of \( M_1 \ M_2 \).

More generally, given a type context \( T[\alpha] \) where \( \alpha \) appears only positively

- \( \forall \alpha. T[\alpha] \) can be instantiated to \( T[\forall \alpha. \alpha] \), and
- \( T[\forall \alpha. \alpha] \) is a subtype of \( \forall \alpha. T[\alpha] \)

Hence, a value of type \( T[\alpha] \) can be given the monomorphic type \( T[\forall \alpha. \alpha] \) by weakening before entering the store to please the value restriction, but retrieved at type \( \forall \alpha. T[\alpha] \), a subtype of \( T[\forall \alpha. \alpha] \).

OCaml implements this, but restricts it to strictly positive occurrences so as to keep the principal type property.
Value restriction

In fact, the two extensions can be combined: \( \Lambda \alpha. M \) need only be *forbidden* when

\( \alpha \) appears in the type of some exposed expansive subterm at some negative occurrence,

where exposed subterms are those that do not appear under some \( \lambda \)-abstraction.

For instance, the expression

\[
\text{let } x : \forall \alpha. \text{int} \times (\text{list } \alpha) \times (\alpha \to \alpha) = \\
\Lambda \alpha. (\text{ref } (1 + 2), (\lambda x : \alpha. x) \text{ Nil}, \lambda x : \alpha. x) \\
in M
\]

may be accepted because \( \alpha \) appears only in the type of the non-expansive exposed expression \( \lambda x : \alpha. x \) and only positively in the type of the expansive expression \( (\lambda x : \alpha. x) \text{ Nil} \).
Conclusions

Experience has shown that *the value restriction is tolerable*. Even though it is not conservative, the search for better solutions has been pretty much abandoned.

There is still ongoing research for tracing side effects more precisely, in particular to better circumvent their use.

Actually, there is a regained interest in tracing side effects, with the introduction of effect handlers.
Conclusions

In a type-and-effect system [Lucassen and Gifford, 1988; Talpin and Jouvelot, 1994], or in a type-and-capability system [Charguéraud and Pottier, 2008], the type system indicates which expressions may allocate new references, and at which type. This permits strong updates—updates that may also change the type of references.

There, the value restriction is no longer necessary.

However, if one extends a type-and-capability system with a mechanism for hiding state, the need for the value restriction re-appears.

Pottier and Protzenko [2012] (and [Protzenko, 2014]) designed a language, called Mezzo, where mutable state is tracked very precisely, using permissions, ownership, and affine types.
Bibliography I

(Most titles have a clickable mark “▷” that links to online versions.)


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.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université Paris 7, June 1972.


Bibliography VI


Bibliography VIII


Bibliography


