MPRI, Typage

Didier Rémy
(With course material from François Pottier)

November 20, 2014
Plan of the course

Introduction

Simply-typed $\lambda$-calculus

Polymorphism and System F

Type reconstruction

Existential types
Existential types
Contents

- Introduction

- Towards typed closure conversion

- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types

- 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.
- 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.
Contents

- Introduction

- Towards typed closure conversion

- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types

- Typed closure conversion
  - Environment passing
  - Closure passing
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* $\lambda$-abstractions, which can have free variables;
- the **target** calculus has *binary* $\lambda$-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 \( \text{fv}(\lambda x. M) \):

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

\[
\begin{align*}
\llbracket M_1 \ M_2 \rrbracket &= \text{let clo} = \llbracket M_1 \rrbracket \text{ in} \\
&\quad \text{let code} = \text{proj}_0 \text{ clo} \text{ in} \\
&\quad \text{code} (\text{clo}, \llbracket M_2 \rrbracket)
\end{align*}
\]

(The variables \textit{code} and \textit{clo} must be suitably fresh.)
Closure-passing closure conversion

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

\[
\llbracket \lambda x. M \rrbracket = \text{let code} = \lambda (\text{clo}, x). \\
\quad \text{let } (\_, x_1, \ldots, x_n) = \text{clo in } \llbracket M \rrbracket \in \\
\quad (\text{code}, x_1, \ldots, x_n)
\]

\[
\llbracket M_1 \ M_2 \rrbracket = \text{let } \text{clo} = \llbracket M_1 \rrbracket \text{ in } \\
\quad \text{let code} = \text{proj}_0 \ \text{clo in} \\
\quad \text{code } (\text{clo}, \llbracket M_2 \rrbracket)
\]

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

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

\[
\begin{align*}
\left[ \lambda x. M \right] &= \text{let code} = \lambda (\text{env}, x). \\
&\quad \text{let } (x_1, \ldots, x_n) = \text{env in } \left[ M \right] \text{ in} \\
&\quad (\text{code}, (x_1, \ldots, x_n))
\end{align*}
\]

\[
\begin{align*}
\left[ M_1 \ M_2 \right] &= \text{let } (\text{code}, \text{env}) = \left[ M_1 \right] \text{ in} \\
&\quad \text{code } (\text{env}, \left[ M_2 \right])
\end{align*}
\]
Environment-passing closure conversion

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

\[
\begin{align*}
[\lambda x. M] &= \text{let code} = \lambda(\text{env}, x). \\
&\hspace{1cm} \text{let} (x_1, \ldots, x_n) = \text{env} \text{ in } [M] \text{ in} \\
&\hspace{1cm} (\text{code}, (x_1, \ldots, x_n)) \\
\end{align*}
\]

\[
\begin{align*}
[M_1 M_2] &= \text{let} (\text{code}, \text{env}) = [M_1] \text{ in} \\
&\hspace{1cm} \text{code} \ (\text{env}, [M_2])
\end{align*}
\]

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

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

\[
\begin{align*}
\llbracket \lambda x. M \rrbracket &= \text{let code} = \lambda (\text{env}, x).
\text{let } (x_1, \ldots, x_n) = \text{env in } \llbracket M \rrbracket \text{ in} \newline
& \quad (\text{code}, (x_1, \ldots, x_n)) \\
\llbracket M_1 \ M_2 \rrbracket &= \text{let } (\text{code}, \text{env}) = \llbracket M_1 \rrbracket \text{ in} \newline
& \quad \text{code } (\text{env}, \llbracket M_2 \rrbracket)
\end{align*}
\]

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 \( \text{fv}(\lambda x. M) \).

\[
\begin{align*}
\llbracket \lambda x. M \rrbracket &= \text{let code} = \lambda (\text{env}, x). \\
& \hspace{1em} \text{let } (x_1, \ldots, x_n) = \text{env} \text{ in } \llbracket M \rrbracket \text{ in} \\
& \hspace{3em} (\text{code}, (x_1, \ldots, x_n))
\end{align*}
\]

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

Write \( \llbracket \Gamma \rrbracket \) for the tuple type \( \tau'_1 \times \ldots \times \tau'_n \) where \( \Gamma \) is \( x_1 : \tau'_1; \ldots; x_n : \tau'_n \).

By hypothesis, we have \( \llbracket \Gamma \rrbracket, x : [\tau_1] \vdash [M] : [\tau_2] \), so

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

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

Can we adopt this as a definition?

\[
[\tau_1 \to \tau_2] = (([\Gamma] \times [\tau_1]) \to [\tau_2]) \times [\Gamma]
\]
Towards a type translation

Can we adopt this as a definition?

\[
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = (\llbracket \Gamma \rrbracket \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket) \times \llbracket \Gamma \rrbracket
\]

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

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

Yes, we do.
Towards a type translation

Yes, we do.

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

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

Yes, we do.

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

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

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

Furthermore, we want function invocations to be translated uniformly, without knowledge of the size and layout of the closure’s environment.
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 \to \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:

$$\text{let } f = \text{let } x = \text{ref } 0 \text{ in } \lambda(). x := (x + 2); ! x$$

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].
Contents

- Introduction

- Towards typed closure conversion

- Existential types
  - Implicitly-type existential types passing
  - Iso-existent types

- Typed closure conversion
  - Environment passing
  - Closure passing
Existential types

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

\[ \tau ::= \ldots \mid \exists \alpha. \tau \]

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

Let’s just 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 \\
\Gamma \vdash \text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2 : \tau_2
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**

\[
\Gamma \vdash M_1 : \exists \alpha. \tau_1 \\
\Gamma, \alpha, x : \tau_1 \vdash M_2 : \tau_2 \\
\Gamma \vdash \text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2 : \tau_2
\]

Anything wrong?
Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

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

\[
\frac{\text{UNPACK}}{
\Gamma \vdash M_1 : \exists \alpha. \tau_1 \\
\Gamma, \alpha, x : \tau_1 \vdash M_2 : \tau_2 \\
\alpha \not\equiv \tau_2 \\
\Gamma \vdash \text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2 : \tau_2}
\]

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

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

Here is how the existential quantifier is introduced and eliminated:

**Pack**

| Γ ⊢ M : [α ↦ τ′]τ | Γ ⊢ pack τ′, M as ∃α. τ : ∃α. τ |

**Unpack**

<table>
<thead>
<tr>
<th>Γ ⊢ M₁ : ∃α. τ₁</th>
<th>Γ ⊢ M₂ : τ₂</th>
<th>α ≠ τ₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ, α, x : τ₁ ⊢ M₂ : τ₂</td>
<td>Γ ⊢ let α, x = unpack M₁ in M₂ : τ₂</td>
<td></td>
</tr>
</tbody>
</table>

The side condition \( α ≠ τ₂ \) is mandatory here to ensure well-formedness of the conclusion.

The side condition may also be written \( Γ ⊢ τ₂ \) which implies \( α ≠ τ₂ \), given that the well-formedness of the last premise implies \( α ∉ \text{dom}(Γ) \).

Note the *imperfect duality* between universals and existentials:

**TAbs**

| Γ, α ⊢ M : τ | Γ ⊢ Λα.M : ∀α. τ |

**TApp**

| Γ ⊢ M : ∀α. τ | Γ ⊢ M τ′ : [α ↦ τ′]τ |
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?
On existential elimination

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

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

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 can 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 $M_2$.

\[
\text{UNPACK} \\
\frac{}{\frac{\Gamma \vdash M_1 : \forall \alpha. \tau_1}{\Gamma, \alpha ; x : \tau_1 \vdash M_2 : \tau_2 \quad \alpha \nmid \tau_2} \quad \Gamma \vdash \text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2 : \tau_2}
\]

The restriction $\alpha \nmid \tau_2$ prevents writing “let $\alpha, x = \text{unpack } M_1 \text{ in } x”$, which would be equivalent to the unsound “\text{unpack } M” of 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 rule could be written

$$\Gamma \vdash M_1 : \exists \alpha. \tau_1 \quad \Gamma \vdash \Lambda \alpha. \lambda x. M_2 : \forall \alpha. \tau_1 \to \tau_2 \quad \alpha \not\equiv \tau_2$$

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

or, more economically:

$$\Gamma \vdash M_1 : \exists \alpha. \tau_1 \quad \Gamma \vdash M_2 : \forall \alpha. \tau_1 \to \tau_2 \quad \alpha \not\equiv \tau_2$$

$$\Gamma \vdash \text{unpack } M_1 \ M_2 : \tau_2$$
On existential elimination

In fact, $M_2$ must be **polymorphic** in $\alpha$: the rule could be written

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

or, more economically:

$$
\frac{
\Gamma \vdash M_1 : \exists \alpha. \tau_1 \\
\Gamma \vdash M_2 : \forall \alpha. \tau_1 \rightarrow \tau_2}{
\Gamma \vdash \text{unpack } M_1 \ M_2 : \tau_2}
$$

One could even view “$\text{unpack}_{\exists \alpha. \tau}$” as a **constant**, equipped with an appropriate type:
On existential elimination

In fact, $M_2$ must be *polymorphic* in $\alpha$: the rule could be written

$$
\Gamma \vdash M_1 : \exists \alpha. \tau_1 \quad \Gamma \vdash \Lambda \alpha. \lambda x. M_2 : \forall \alpha. \tau_1 \rightarrow \tau_2 \quad \alpha \not\equiv \tau_2
$$

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

or, more economically:

$$
\Gamma \vdash M_1 : \exists \alpha. \tau_1 \quad \Gamma \vdash M_2 : \forall \alpha. \tau_1 \rightarrow \tau_2 \quad \alpha \not\equiv \tau_2
$$

$$
\Gamma \vdash \text{unpack } M_1 \ M_2 : \tau_2
$$

One could even view “unpack$_{\exists \alpha. \tau}$” as a *constant*, equipped with an appropriate type:

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

The variable $\beta$, which stands for $\tau_2$, is bound prior to $\alpha$, so it naturally cannot be instantiated to a type that refers to $\alpha$. This reflects the side condition $\alpha \not\equiv \tau_2$. 
On existential introduction

\[
\text{PACK} \quad \Gamma \vdash M : [\alpha \mapsto \tau']\tau \\
\hline
\Gamma \vdash \text{pack} \tau', M \text{ as } \exists\alpha. \tau : \exists\alpha.\tau
\]
Existentials as constants

In System F, existential types can also be presented as constants

\[
\begin{align*}
\text{pack}_{\exists \alpha. \tau} & : \forall \alpha. (\tau \to \exists \alpha. \tau) \\
\text{unpack}_{\exists \alpha. \tau} & : \exists \alpha. \tau \to \forall \beta. ((\forall \alpha. (\tau \to \beta)) \to \beta)
\end{align*}
\]

Read:

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

\[
\llbracket \exists \alpha. \tau \rrbracket = \forall \beta. ((\forall \alpha. (\llbracket \tau \rrbracket \to \beta)) \to \beta) \quad \text{if } \beta \not= \tau
\]

The term translation is:

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

\[
\llbracket \text{unpack}_{\exists \alpha. \tau} \rrbracket : \llbracket \exists \alpha. \tau \rrbracket \to \forall \beta. ((\forall \alpha. (\llbracket \tau \rrbracket \to \beta)) \to \beta) \\
= ?
\]
Encoding existentials into universals

The type translation is *double negation*:

\[
[\exists \alpha. \tau] = \forall \beta.((\forall \alpha.([\tau] \to \beta)) \to \beta) \quad \text{if } \beta \neq \tau
\]

The term translation is:

\[
\begin{align*}
\llbracket \text{pack}_{\exists \alpha. \tau} \rrbracket & : \forall \alpha.([\tau] \to [\exists \alpha. \tau]) \\
& = \Lambda \alpha. \lambda x : [\tau]. \Lambda \beta. \lambda k : \forall \alpha.([\tau] \to \beta). ? : \beta \\
\llbracket \text{unpack}_{\exists \alpha. \tau} \rrbracket & : [\exists \alpha. \tau] \to \forall \beta.((\forall \alpha.([\tau] \to \beta)) \to \beta) \\
& = ?
\end{align*}
\]
Encoding existentials into universals

The type translation is *double negation*:

$$\downbracket{\exists \alpha. \tau} = \forall \beta. ((\forall \alpha. (\downbracket{\tau} \rightarrow \beta)) \rightarrow \beta) \quad \text{if } \beta \neq \tau$$

The term translation is:

$$\downbracket{\text{pack}_{\exists \alpha. \tau}} : \forall \alpha. (\downbracket{\tau} \rightarrow \downbracket{\exists \alpha. \tau})$$

$$= \Lambda \alpha. \lambda x : \downbracket{\tau}. \Lambda \beta. \lambda k : \forall \alpha. (\downbracket{\tau} \rightarrow \beta). k \alpha x$$

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

$$= ?$$
Encoding existentials into universals

The type translation is *double negation*:

\[
[\exists \alpha. \tau] = \forall \beta. ((\forall \alpha. ([\tau] \rightarrow \beta)) \rightarrow \beta) \quad \text{if } \beta \neq \tau
\]

The term translation is:

\[
\begin{align*}
\text{[pack}_{\exists \alpha. \tau}] & : \quad \forall \alpha. ([\tau] \rightarrow [\exists \alpha. \tau]) \\
& = \Lambda \alpha. \lambda x : [\tau]. \Lambda \beta. \lambda k : \forall \alpha. ([\tau] \rightarrow \beta). k \alpha x \\
\text{[unpack}_{\exists \alpha. \tau}] & : \quad [\exists \alpha. \tau] \rightarrow \forall \beta. ((\forall \alpha. ([\tau] \rightarrow \beta)) \rightarrow \beta) \\
& = \lambda x : [\exists \alpha. \tau]. x
\end{align*}
\]

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

What is the computational content of this encoding?
Encoding existentials into universals

The type translation is \textit{double negation}:

\[
\llbracket \exists \alpha. \tau \rrbracket = \forall \beta. ((\forall \alpha. (\llbracket \tau \rrbracket \rightarrow \beta)) \rightarrow \beta) \quad \text{if } \beta \neq \tau
\]

The term translation is:

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

\[
\quad = \Lambda \alpha. \lambda x : \llbracket \tau \rrbracket. \Lambda \beta. \lambda k : \forall \alpha. (\llbracket \tau \rrbracket \rightarrow \beta). k \alpha x
\]

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

\[
\quad = \lambda x : \llbracket \exists \alpha. \tau \rrbracket. x
\]

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 as constants

\( \text{pack}_{\exists \alpha. \tau} \) can be treated as a unary constructor, and \( \text{unpack}_{\exists \alpha. \tau} \) as a unary destructor. The \( \delta \)-reduction rule is:

\[
\text{unpack}_{\exists \alpha. \tau_0} (\text{pack}_{\exists \alpha. \tau} \tau' V) \rightarrow \Lambda \beta. \lambda y : \forall \alpha. \tau \rightarrow \beta . y \, \tau' \, V
\]

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

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

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 extended.
The semantics of existential types as primitive

We extend values and evaluation contexts as follows:

\[
V ::= \ldots \text{pack } \tau', V \text{ as } \tau
\]

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

We add the reduction rule:

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

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* \( \text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2 \) *could be problematic when* \( M_1 \) *is not a value.*
The semantics of existential types

Exercise

Find an example that illustrates why the reduction of
let α, x = unpack M₁ in M₂ could be problematic when M₁ is not a value.

Need a hint?

Use a conditional
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.

Solution

Let $M_1$ be if $M$ then $V_1$ else $V_2$ where $V_i$ is of the form $\text{pack } \tau_i, V_i \text{ as } \exists \alpha. \tau$ 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?
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.
- Introduction

- Towards typed closure conversion

- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types

- 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:

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

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

The semantics of this let-binding is as before:

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

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

Yes, it is.

But there is a subtlety!
Implicitly-typed existential types

Yes, it is.

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

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

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 $bool$. 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 $bool \to \tau_0$.

$$a_1 = \text{let } x = v \ v_0 \text{ in } x \ (x \ (\lambda y. y)) \quad \rightarrow \quad v \ v_0 \ (v \ v_0 \ (\lambda y. y)) = a_2$$

We have $\emptyset \vdash a_1 : \exists \alpha. \alpha \to \alpha$ while $\emptyset \nmid a_2 : \tau$.

What happened?
Implicitly-typed existential types

Let $\tau_0$ be $\exists \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha)$ and $v_0$ a value of type $bool$. Let $v_1$ and $v_2$ be two values of type $\tau_0$ 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 $bool \to \tau_0$.

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

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

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?
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:

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

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

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 break.
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):

\[
\left[\text{unpack } a_1 \ (\lambda x. a_2)\right] = \left[a_1\right] (\lambda x. \left[a_2\right])
\] (1)

\[
\rightarrow (\lambda k. \left[a\right] k) (\lambda x. \left[a_2\right])
\] (2)

\[
\rightarrow (\lambda x. \left[a_2\right]) \left[a\right]
\] (3)

\[
\rightarrow [x \mapsto \left[a\right]] \left[a_2\right]
\] (4)

In the call-by-value setting, \(\lambda k. \left[a\right] k\) would come from the reduction of \(\left[pack a\right]\), i.e. is \((\lambda k. \lambda x. k \ x) \left[a\right]\), 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 \(\lambda k. \left[a\right] k\).
• Introduction

• Towards typed closure conversion

• Existential types
  • Implicitly-type existential types passing
  • Iso-existential types

• 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 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*:

\[ D \tilde{\alpha} \approx \exists \tilde{\beta}. \tau \quad \text{if } \text{ftv}(\tau) \subseteq \tilde{\alpha} \cup \tilde{\beta} \quad \text{and} \quad \tilde{\alpha} \not\equiv \tilde{\beta} \]

This introduces two constants, with the following type schemes:

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

(Compare with basic iso-recursive types, where $\tilde{\beta} = \emptyset$.)
Iso-existential types in ML

A few corners have been cut on the previous slide. The “type scheme:”

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

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

A few corners have been cut on the previous slide. The “type scheme:"

\[ \forall \vec{\alpha} \gamma. D \vec{\alpha} \rightarrow (\forall \vec{\beta}. (\tau \rightarrow \gamma)) \rightarrow \gamma \]

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

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

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

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

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

We have seen a version of this rule in System F earlier; this in an ML version. The term \( M_2 \) must be polymorphic, which \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):

$$\langle \text{unpack}_D \, M_1 \, M_2 : \tau_2 \rangle = \exists \alpha. \left( \langle M_1 : D \, \alpha \rangle \forall \beta. \langle M_2 : \tau \to \tau_2 \rangle \right)$$

where $D \, \alpha \approx \exists \beta. \tau$ and, w.l.o.g., $\alpha \beta \not\equiv M_1, M_2, \tau_2$.

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 (somewhat bizarre) syntax for this in OCaml is:

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

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

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

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

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

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

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

Define

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

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

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

**Exercise** We reuse the type $D \alpha \approx \exists \beta. (\beta \rightarrow \alpha) \times \beta$ of frozen computations. Assume given a list $l$ with elements of type $D \tau_1$. Assume given a function $g$ of type $\tau_1 \rightarrow \tau_2$. Transform the list into a new list $l'$ of frozen computations of type $D \tau_2$ (without actually running any computation).

$$\text{List.map } (\lambda (z) \text{ let } D(f, y) = z \text{ in } D((\lambda (z) g (f z)), y))$$

Generalizing this example to a function that receives $g$ and $l$ and returns $l'$ does not typecheck:

$$\text{let lift g l =}$$

$$\text{List.map } (\lambda (z) \text{ let } D(f, y) = z \text{ in } D((\lambda (z) g (f z)), y))$$

In expression `let $\alpha, x = \text{unpack } M_1 \text{ in } M_2$, occurrences of $x$ in $M_2$ can only be passed to external functions (free variables) that are polymorphic so that $x$ does not leak out of its context.
Uses of existential types

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

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

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.
**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)**

```ml
type 'a d = D : ('b -> 'a) * 'b -> 'a d
let freeze f x = D (f, x)
let run (D (f, x)) = f x
```

**Via first-class modules (abstract types)**

```ml
module type D = sig type b type a val f : b -> a val x : b end
let freeze (type u) (type v) f x =
    (module struct type b = u type a = v let f = f let x = x end : D);;
let unfreeze (type u) (module M : D with type a = u) = M.f M.x
```
**Contents**

- **Introduction**
- **Towards typed closure conversion**
- **Existential types**
  - Implicitly-type existential types passing
  - Iso-existential types
  - **Typed closure conversion**
    - Environment passing
    - Closure passing
Introduction

Towards typed closure conversion

Existential types
  - Implicitly-type existential types passing
  - Iso-existential types

Typed closure conversion
  - Environment passing
  - Closure passing
Typed closure conversion

Everything is now set up to prove that, in System F with existential types:

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

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

$$\begin{align*}
\llbracket \lambda x : \tau_1. M \rrbracket &= \text{let code : } \\
&= \lambda (env : \ldots, x : \ldots). \\
&\quad \text{let } (x_1, \ldots, x_n : \ldots) = env \text{ in} \\
&\quad \llbracket M \rrbracket \\
&\quad \text{in} \\
&\quad \text{pack } (\text{code}, (x_1, \ldots, x_n)) \\
&\quad \text{as}
\end{align*}$$

We find $\llbracket \Gamma \rrbracket \vdash \llbracket \lambda x : \tau_1. M \rrbracket : \llbracket \tau_1 \to \tau_2 \rrbracket$, as desired.
Environment-passing closure conversion

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

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

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

$$[[M]]$$

$$\text{in}$$

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

$$\text{as}$$

We find $[[\Gamma]] \vdash [[\lambda x : \tau_1. M]] : [[\tau_1 \rightarrow \tau_2]]$, as desired.
Towards typed closure conversion

Environment-passing closure conversion

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

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

$$\text{let (}x_1, \ldots, x_n : \llbracket \Gamma \rrbracket\text{) = env in}$$

$$\llbracket M \rrbracket \text{ in pack , (code , (}x_1, \ldots, x_n\text{)) as}$$

We find $\llbracket \Gamma \rrbracket \vdash \llbracket \lambda x : \tau_1 . M \rrbracket : \llbracket \tau_1 \rightarrow \tau_2 \rrbracket$, as desired.
Environment-passing closure conversion

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

$$\left[ \lambda x : \tau_1 . M \right] = \text{let } \text{code} : (\left[\Gamma\right] \times \left[\tau_1\right]) \to \left[\tau_2\right] = \lambda (\text{env} : \left[\Gamma\right], x : \left[\tau_1\right]). \text{let } (x_1, \ldots, x_n : \left[\Gamma\right]) = \text{env in} \left[ M \right] \text{ in} \text{pack } \left[\Gamma\right], (\text{code}, (x_1, \ldots, x_n)) \text{ as } \exists \alpha.((\alpha \times \left[\tau_1\right]) \to \left[\tau_2\right]) \times \alpha$$

We find $\left[\Gamma\right] \vdash \left[ \lambda x : \tau_1 . M \right] : \left[\tau_1 \to \tau_2\right]$, as desired.
Environment-passing closure conversion

Assume $\Gamma \vdash M : \tau_1 \to \tau_2$ and $\Gamma \vdash M_1 : \tau_1$. 

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

$$\text{code } (\text{env}, [[M_1]])$$

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

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

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?
Recursive functions can be translated in this way, known as the “fix-code” variant [Morrisett and Harper, 1998]:

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

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.
Environment-passing closure conversion

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

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

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

This requires general, recursively-defined \textit{values}. Closures are now \textit{cyclic} data structures.
Environment-passing closure conversion

Here is how the “fix-pack” variant is type-checked. Assume
\[ \Gamma \vdash \mu f. \lambda x. M : \tau_1 \to \tau_2 \quad \text{and} \quad \text{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \text{fv}(\mu f. \lambda x. M). \]

\[
\llbracket \mu f \ . \lambda x. M \rrbracket = \\
\begin{aligned}
\text{let } code : & = \\
\lambda (env : & , x : ) . \\
\text{let } (f, x_1, \ldots, x_n) : & = env \ in \\
\text{let rec } clo : & = \\
pack & , (code, (clo, x_1, \ldots, x_n)) \\
as & in clo
\end{aligned}
\]
Environment-passing closure conversion

Here is how the “fix-pack” variant is type-checked. Assume
\[ \Gamma \vdash \mu f. \lambda x. M : \tau_1 \rightarrow \tau_2 \] and \( \text{dom}(\Gamma) = \{ x_1, \ldots, x_n \} = \text{fv}(\mu f. \lambda x. M) \).

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

\[
\text{let code : } \lambda (\text{env} : \llbracket f : \tau_1 \rightarrow \tau_2, \Gamma \rrbracket, x : \llbracket \tau_1 \rrbracket). \]

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

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

\[
\text{let rec clo : } \text{pack } \llbracket f : \tau_1 \rightarrow \tau_2, \Gamma \rrbracket, (\text{code}, (\text{clo}, x_1, \ldots, x_n)) \text{ as}
\]

\[
in \text{clo}
\]
Here is how the “fix-pack” variant is type-checked. Assume
\( \Gamma \vdash \mu f. \lambda x. M : \tau_1 \to \tau_2 \) and \( \text{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \text{fv}(\mu f. \lambda x. M) \).

\[
\left[ \mu f : \tau_1 \to \tau_2. \lambda x. M \right] = \\
\lambda \left( \text{env} : \left[ f : \tau_1 \to \tau_2, \Gamma \right] \times \left[ \tau_1 \right] \right). \\
\text{let } code : \left( \left[ f : \tau_1 \to \tau_2; \Gamma \right] \times \left[ \tau_1 \right] \right) \to \left[ \tau_2 \right] = \\
\text{let } (f, x_1, \ldots, x_n) : \left[ f : \tau_1 \to \tau_2, \Gamma \right] = \text{env } \text{in} \\
\left[ M \right] \text{ in} \\
\text{let rec } clo : \\
\text{pack } \left[ f : \tau_1 \to \tau_2, \Gamma \right], (\text{code}, (\text{clo}, x_1, \ldots, x_n)) \\
as \\
in \text{clo}
Here is how the “fix-pack” variant is type-checked. Assume
\[ \Gamma \vdash \mu f. \lambda x. M : \tau_1 \rightarrow \tau_2 \text{ and } \text{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \text{fv}(\mu f. \lambda x. M). \]

\[
\llbracket \mu f : \tau_1 \rightarrow \tau_2, \lambda x. M \rrbracket = \\
\text{let } \text{code} : (\llbracket f : \tau_1 \rightarrow \tau_2; \Gamma \rrbracket \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket = \\
\lambda (env : \llbracket f : \tau_1 \rightarrow \tau_2, \Gamma \rrbracket, x : \llbracket \tau_1 \rrbracket). \\
\text{let } (f, x_1, \ldots, x_n) : \llbracket f : \tau_1 \rightarrow \tau_2, \Gamma \rrbracket = env \text{ in } \\
\llbracket M \rrbracket \text{ in } \\
\text{let rec } \text{clo} : \llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \\
\text{pack } \llbracket f : \tau_1 \rightarrow \tau_2, \Gamma \rrbracket, (\text{code}, (\text{clo}, x_1, \ldots, x_n)) \text{ as } \\
in \text{clo}
\]
Here is how the “fix-pack” variant is type-checked. Assume
Γ ⊢ μf.λx.M : τ₁ → τ₂ and dom(Γ) = {x₁, ..., xₙ} = fv(μf.λx.M).

\[
\begin{align*}
\llbracket μf : τ₁ → τ₂, λx.M \rrbracket &= \\
let code : (\llbracket f : τ₁ → τ₂; Γ \rrbracket \times [τ₁]) \to [τ₂] &= \\
\lambda (env : [f : τ₁ → τ₂, Γ], x : [τ₁]). \\
let (f, x₁, ..., xₙ) : [f : τ₁ → τ₂, Γ] = env in \\
\llbracket M \rrbracket in \\
let rec clo : [τ₁ → τ₂] = \\
pack [f : τ₁ → τ₂, Γ], (code, (clo, x₁, ..., xₙ)) \\
as 3∃α((α \times [τ₁]) \to [τ₂]) \times α in clo 
\end{align*}
\]
Environment-passing closure conversion

Here is how the “fix-pack” variant is type-checked. Assume
\( \Gamma \vdash \mu f. \lambda x. M : \tau_1 \rightarrow \tau_2 \) and \( \text{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \text{fv}(\mu f. \lambda x. M) \).

\[
\llbracket \mu f : \tau_1 \rightarrow \tau_2, \lambda x. M \rrbracket = \\
\begin{align*}
\text{let } & \text{code} : (\llbracket f : \tau_1 \rightarrow \tau_2; \Gamma \rrbracket \times [\tau_1]) \rightarrow [\tau_2] = \\
& \lambda (\text{env} : \llbracket f : \tau_1 \rightarrow \tau_2, \Gamma \rrbracket, x : [\tau_1]). \\
\text{let } & (f, x_1, \ldots, x_n) : \llbracket f : \tau_1 \rightarrow \tau_2, \Gamma \rrbracket = \text{env} \text{ in } \\
\llbracket M \rrbracket \text{ in } \\
\text{let rec } & \text{clo} : [\tau_1 \rightarrow \tau_2] = \\
& \text{pack } [\llbracket f : \tau_1 \rightarrow \tau_2, \Gamma \rrbracket, (\text{code}, (\text{clo}, x_1, \ldots, x_n)) \\
& \text{as } \exists \alpha((\alpha \times [\tau_1]) \rightarrow [\tau_2]) \times \alpha \\
& \text{in } \text{clo}
\end{align*}
\]

Problem?
Environment-passing closure conversion

The recursive function may be polymorphic, but recursive calls are monomorphic...

We can generalize the encoding afterwards,

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

whenever the right-hand side is well-defined.

This allows the *indirect* compilation of polymorphic recursive functions as long as the recursion is monomorphic.

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

\[
\llbracket \mu f : \forall \vec{\beta}. \tau_1 \to \tau_2. \lambda x. M \rrbracket = \\
\text{let code : } \forall \vec{\beta}. (\llbracket f : \forall \vec{\beta}. \tau_1 \to \tau_2; \Gamma \rrbracket \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket = \\
\lambda (env : \llbracket f : \forall \vec{\beta}. \tau_1 \to \tau_2, \Gamma \rrbracket, x : \llbracket \tau_1 \rrbracket). \\
\text{let } (f, x_1, \ldots, x_n) : \llbracket f : \forall \vec{\beta}. \tau_1 \to \tau_2, \Gamma \rrbracket = env \text{ in} \\
\llbracket M \rrbracket \text{ in} \\
\text{let rec clo : } \llbracket \forall \vec{\beta}. \tau_1 \to \tau_2 \rrbracket = \\
\Lambda \vec{\beta}. \text{pack } \llbracket f : \forall \vec{\beta}. \tau_1 \to \tau_2, \Gamma \rrbracket, (code \vec{\beta}, (clo, x_1, \ldots, x_n)) \\
as \exists \alpha ((\alpha \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket) \times \alpha) \\
in clo
\]

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.
Introduction

Towards typed closure conversion

Existential types
  - Implicitly-type existential types passing
  - Iso-existential types

Typed closure conversion
  - Environment passing
  - Closure passing
Closure-passing closure conversion

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

\[
\left[ M_1 \ M_2 \right] = \begin{array}{l}
\text{let } \text{clo} = \left[ M_1 \right] \text{ in} \\
\quad \text{let } \text{code} = \text{proj}_0 \text{clo} \text{ in} \\
\quad \text{code} (\text{clo}, \left[ M_2 \right])
\end{array}
\]

where \( \{ x_1, \ldots, x_n \} = \text{fv}(\lambda x. M) \).
Closure-passing closure conversion

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

\[
\left[ M_1 \ M_2 \right] = \begin{array}{l}
\text{let } \text{clo} = \left[ M_1 \right] \text{ in}
\text{let } \text{code} = \text{proj}_{0} \text{clo in}
\text{code } (\text{clo}, \left[ M_2 \right])
\end{array}
\]

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

How could we typecheck this? What are the difficulties?
Closure-passing closure conversion

\[
\begin{align*}
\llbracket \lambda x. M \rrbracket & = \text{let code} = \lambda (\text{clo}, x). \\
& \hspace{1em} \text{let } (\_, x_1, \ldots, x_n) = \text{clo} \text{ in} \\
& \hspace{2em} \llbracket M \rrbracket \\
& \hspace{3em} \text{in } (\text{code}, x_1, \ldots, x_n) \\
\llbracket M_1 \ M_2 \rrbracket & = \text{let clo} = \llbracket M_1 \rrbracket \text{ in} \\
& \hspace{1em} \text{let code} = \text{proj}_0 \text{ clo in} \\
& \hspace{2em} \text{code} (\text{clo}, \llbracket M_2 \rrbracket)
\end{align*}
\]

There are two difficulties:

- a closure is a tuple, whose first field should be exposed (it is the code pointer), while the number and types of the remaining fields should be abstract;
- the first field of the closure contains a function that expects the closure itself as its first argument.
Closure-passing closure conversion

There are two difficulties:

- a closure is a tuple, whose *first* field should be *exposed* (it is the code pointer), while the number and types of the remaining fields should be abstract;
- the first field of the closure contains a function that expects *the closure itself* as its first argument.

What type-theoretic mechanisms could we use to describe this?
Closure-passing closure conversion

There are two difficulties:

- a closure is a tuple, whose *first* field should be *exposed* (it is the code pointer), while the number and types of the remaining fields should be abstract;
- the first field of the closure contains a function that expects *the closure itself* as its first argument.

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

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

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

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

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:

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

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

These rules make sense with or without row variables.

Projection does not care about the fields beyond \(i\). Thanks to row variables, this can be expressed in terms of \textit{parametric polymorphism}:

\[
\text{proj}_i : \forall \alpha_1 \ldots \alpha_i \rho. \Pi (\alpha_1; \ldots; \alpha_i; \rho) \rightarrow \alpha_i
\]
About Rows

Rows were invented by Wand and improved by Rémy 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 Rémy [Pottier and Rémy, 2005].
Closure-passing closure conversion

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

\[
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \exists \rho. \mu \alpha. \Pi \left( (\alpha \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket; \rho \right)
\]

\(\rho\) 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].
Closure-passing closure conversion

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

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \exists \rho. \mu\alpha. \Pi (\alpha \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket; \rho$$

$\rho$ 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 $\mu\alpha. \exists \rho. \tau$
Closure-passing closure conversion

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

\[
\llbracket \tau_1 \to \tau_2 \rrbracket = \exists \rho. \mu \alpha. \Pi (\alpha \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket; \\
\rho \text{ describes the environment} \\
\alpha \text{ is the concrete type of the closure} \\
\text{a tuple...} \\
\text{...that begins with a code pointer...} \\
\text{...and continues with the environment}
\]

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.
Closure-passing closure conversion

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

\[
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \exists \rho. \quad \mu \alpha. \quad \Pi ( (\alpha \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket; \rho)
\]

\(\rho\) 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:** Notice that \(\rho\) appears only once. Any comments?
Rows and recursive types allow to define the translation of types in the closure-passing variant:

\[
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \exists \rho. \\
\mu \alpha. \\
\Pi ( \\
(\alpha \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket); \\
\rho
\]

ρ describes the environment
α 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:** Notice that ρ appears only once. Any comments?

*Usually, an existential type variable appears both at positive and negative occurrences.*
Closure-passing closure conversion

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

\[
\llbracket \tau_1 \to \tau_2 \rrbracket = \exists \rho. \\
\mu \alpha. \\
\Pi ( (\alpha \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket; \rho)
\]

\(\rho\) 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:** Notice that \(\rho\) appears only once. Any comments?

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

Let \( \text{Clo}(R) \) abbreviate \( \mu \alpha. \Pi ((\alpha \times [\tau_1]) \to [\tau_2]; R) \).

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

We have \([\tau_1 \to \tau_2] = \exists \rho. \text{Clo}(\rho)\).

\[
\begin{align*}
[\lambda x : \cdot . M] & = \text{let code : } \lambda (\, \text{clo} : \cdot \cdot , x : \cdot) . \ \\
& \text{let } (\_ , x_1 , \ldots , x_n) : \ \\
& \text{pack } \langle M \rangle \text{ in } \ \\
& \text{unfold clo in } \ \\
& \text{as } \ \\
\end{align*}
\]

\[
\begin{align*}
[M_1 M_2] & = \text{let } \rho , \, \text{clo} = \text{unpack } [M_1] \text{ in } \ \\
& \text{let code : } \ \\
& \text{proj}_0 (\text{unfold clo}) \text{ in } \ \\
& \text{code (clo, } [M_2] ) \ \\
\end{align*}
\]
Closure-passing closure conversion

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

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

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

$$
\left[ \lambda x : [\tau_1]. M \right] = \text{let code : (} Clo([\Gamma]) \times [\tau_1]\text{) } \to [\tau_2] = \lambda (clo : Clo([\Gamma]), x : [\tau_1]).
\text{let } (_, x_1, \ldots, x_n) : UClo[\Gamma] = \text{unfold clo in}
\left[ M \right] \text{ in pack } [\Gamma], (\text{fold } (\text{code, } x_1, \ldots, x_n)) \text{ as } \exists \rho. Clo(\rho)
$$

$$
\left[ M_1 \ M_2 \right] = \text{let } \rho, \ clo = \text{unpack } [M_1] \text{ in}
\text{let code : (} Clo(\rho) \times [\tau_1]\text{) } \to [\tau_2] = \text{proj}_0 (\text{unfold clo) in}
\text{code } (\text{clo, } [M_2])
$$
Closure-passing closure conversion

In the closure-passing variant, recursive functions can be translated as:

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

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

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

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

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

$\text{unfold } \text{clo in}$

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

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

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

$\text{in } \text{clo}$

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

Here, a polymorphic recursive function is directly compiled into a polymorphic recursive closure. Notice that the type of closures is unchanged, so the encoding of applications is also unchanged.
Mutually recursive functions

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:

\[ \llbracket M \rrbracket = \]

Environment passing:
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:

\[
\llbracket M \rrbracket = \text{let } code_i = \lambda (env, x). \\
\text{let } (f_1, f_2, x_1, \ldots, x_n) = env \text{ in } \\
\llbracket M_i \rrbracket \\
\text{in } \\
\text{let } rec \ clo_1 = (\text{code}_1, (\text{clo}_1, \text{clo}_2, x_1, \ldots, x_n)) \\
\text{and } clo_2 = (\text{code}_2, (\text{clo}_1, \text{clo}_2, x_1, \ldots, x_n)) \text{ in } \\
clo_1, clo_2
\]
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:

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

Comments?
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:

\[
\begin{align*}
&[[M]] = \text{let } \text{code}_i = \lambda (env, x). \\
&\quad \text{let } (f_1, f_2, x_1, \ldots, x_n) = env \text{ in} \\
&\quad [[M_i]] \\
&\quad \text{in} \\
&\quad \text{let rec } env = (\text{clo}_1, \text{clo}_2, x_1, \ldots, x_n) \\
&\quad \text{and } \text{clo}_1 = (\text{code}_1, env) \\
&\quad \text{and } \text{clo}_2 = (\text{code}_2, env) \text{ in} \\
&\quad \text{clo}_1, \text{clo}_2
\end{align*}
\]
Mutually recursive functions

Can we compile mutually recursive functions?

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

Closure passing:

\[
\begin{align*}
& \text{let } code_i = \lambda (clos, x). \\
& \text{let } (\_ , f_1, f_2, x_1, \ldots, x_n) = clos \text{ in } \boxed{M_i} \\
& \text{in} \\
& \text{let rec } clos_1 = (code_1, clos_1, clos_2, x_1, \ldots, x_n) \\
& \text{and } clos_2 = (code_2, clos_1, clos_2, x_1, \ldots, x_n) \\
& \text{in } clos_1, clos_2
\end{align*}
\]
Mutually recursive functions

Can we compile mutually recursive functions?

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

Closure passing:

```
let code_i = \lambda (clo, x).
  let (_, f_1, f_2, x_1, \ldots, x_n) = clo in \llbracket M_i \rrbracket
in
let rec clo_1 = (code_1, clo_1, clo_2, x_1, \ldots, x_n)
  and clo_2 = (code_2, clo_1, clo_2, x_1, \ldots, x_n)
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?

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

Closure passing:

\[
\begin{align*}
\text{let } code_1 &= \lambda (clo, x). \\
& \quad \text{let } (_\text{code}_1, _\text{code}_2, f_1, f_2, x_1, \ldots, x_n) = clo \text{ in } [M_1] \text{ in} \\
\text{let } code_2 &= \lambda (clo, x). \\
& \quad \text{let } (_\text{code}_2, f_1, f_2, x_1, \ldots, x_n) = clo \text{ in } [M_2] \text{ in} \\
\text{let rec } clo_1 &= (code_1, code_2, clo_1, clo_2, x_1, \ldots, x_n) \text{ and } clo_2 = c_1.\text{tail} \\
& \quad \text{in } clo_1, clo_2
\end{align*}
\]

- \textit{\text{clo}_1.\text{tail}} returns a pointer to the tail \((\text{code}_2, \text{clo}_1, \text{clo}_2, x_1, \ldots, x_n)\) of \text{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?
Optimizing representations

Can closure passing and environment passing be mixed?

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

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

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

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

Applications? When many definitions share the same closure, the closure (or part of it) may be shared.
Can closure passing and environment passing be mixed?

No because the calling-convention (i.e., 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:

\[
\begin{align*}
\textstyle \lambda x. M & \quad = \quad \text{let } \text{code} = \lambda (\text{clo}, x). \\
& \quad \quad \text{let } (\_, (x_1, \ldots, x_n)) = \text{clo in } [M] \text{ in } \\
& \quad \quad \quad (\text{code, (}x_1, \ldots, x_n\text{})) \\
\textstyle [M_1 \ M_2] & \quad = \quad \text{let } \text{clo} = [M_1] \text{ in } \\
& \quad \quad \text{let } \text{code} = \text{proj}_0 \text{clo in } \\
& \quad \quad \quad \text{code (}\text{clo, [}M_2\text{]})
\end{align*}
\]

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:

```
class c (x_1, \ldots x_q) {
    meth m_1 = M_1 
    \ldots 
    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:

\[
\begin{align*}
let\ m &= \\
let\ m_1 &= \lambda(m, x_1, \ldots, x_q). M_1 \ in \\
&\quad \ldots \\
let\ m_p &= \lambda(m, x_1, \ldots, x_q). M_p \ in \\
\{m_1, \ldots, m_p\} \ in \\
\lambda x_1 \ldots x_q. (m, x_1, \ldots x_q)
\end{align*}
\]

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 \( C\text{lo}(R) \) be \( \mu \alpha. \Pi(\{(m_i : \alpha \to \tau_i)_{i \in 1..n}\}; R) \) and \( U\text{Clo}(R) \) its unfolding.

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

\[
\text{let } m = \{ \\
\quad m_1 = \lambda(m, x_1, \ldots, x_q : U\text{Clo}(R)). [M_1] \\
\quad \ldots \\
\quad m_p = \lambda(m, x_1, \ldots, x_q : U\text{Clo}(R)). [M_p] \\
\} \text{ in } \\
\lambda x_1. \ldots \lambda x_q. \text{pack } R, \text{fold } (m, x_1, \ldots, x_q) \text{ as } \exists \rho. (M, \rho)
\]

Calling a method of an object \( p \) of type \( M \) is

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

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 \textit{fun}. (Yes, really!)

It forces compiler writers to make the structure of the compiled program \textit{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 naïve 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].
(Most titles have a clickable mark “▷” that links to online versions.)


Bibliography V
