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

Didier Rémy

October 27, 2017

INRIA

Plan of the course

Metatheory of System F

ADTs, Existential types, GATDs

Abstract Data types, Existential types, GADTs

Contents

Type-preserving compilation

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

Why *preserve types* during compilation?

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

Type-preserving compilation

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

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

For example:

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

Type-preserving compilation

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

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

Translating types

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

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

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

See the old lecture on type closure conversion.

Closure conversion

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

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

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

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

Source and target

In the following,

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

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

Variants of closure conversion

There are at least two variants of closure conversion:

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

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

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

Closure-passing closure conversion

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

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

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

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

Environment-passing closure conversion

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

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

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

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

Environment-passing closure conversion

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

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

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

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

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

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

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

Towards a type translation

Can we adopt this as a definition?

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

Naturally not. This definition is mathematically ill-formed: we cannot use Γ out of the blue.

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

Towards a type translation

Yes, we do.

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

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

$$\textit{if} \ \ldots \ \textit{then} \ \lambda x.\, x + y \ \textit{else} \ \lambda x.\, x$$

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

So, *what could be the definition of* $[\![\tau_1 \rightarrow \tau_2]\!]$*?*

The type translation

The only sensible solution is:

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

An *existential quantification* over the type of the environment abstracts away the differences in size and layout.

Enough information is retained to ensure that the application of the code to the environment is valid: this is expressed by letting the variable α occur twice on the right-hand side.

The type translation

The existential quantification also provides a form of *security*: the caller cannot do anything with the environment except pass it as an argument to the code; in particular, it cannot inspect or modify the environment.

For instance, in the source language, the following coding style guarantees that x remains even, no matter how f is used:

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

After closure conversion, the reference x is reachable via the closure of f. A malicious, untyped client could write an odd value to x. However, a *well-typed* client is unable to do so.

This encoding is not just type-preserving, but also *fully abstract:* it preserves (a typed version of) observational equivalence [Ahmed and Blume, 2008].

Contents

- Typed closure conversion

- Typed closure conversion
  - Environment passing
  - Closure passing

Typed closure conversion

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

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

Environment-passing closure conversion

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

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

We find $[\![\Gamma]\!] \vdash [\![\lambda x{:}\tau_1.\, M]\!] : [\![\tau_1 \to \tau_2]\!]$, as desired.

Environment-passing closure conversion

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

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

We find $\llbracket \Gamma \rrbracket \vdash \llbracket M\ M_1 \rrbracket : \llbracket \tau_2 \rrbracket$, as desired.

Environment-passing closure conversion recursion

Recursive functions can be translated in this way, known as the “fix-code” variant [Morrisett and Harper, 1998] (leaving out type information):

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

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

The translation of applications is unchanged: recursive and non-recursive functions have an identical calling convention.

What is the weak point of this variant?

A new closure is allocated at every call.

Environment-passing closure conversion recursion

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

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

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

This requires general, recursively-defined *values.* Closures are now *cyclic* data structures.

Environment-passing closure conversion recursion

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

```
[[μf : τ₁ → τ₂.λx.M]] =
            let code : ([[f : τ₁ → τ₂; Γ]] × [[τ₁]]) → [[τ₂]] =
              λ(env : [[f : τ₁ → τ₂, Γ]], x : [[τ₁]]).
                let (f, x₁, ..., xₙ) : [[f : τ₁ → τ₂, Γ]] = env in
                [[M]] in
            let rec clo : [[τ₁ → τ₂]] =
              pack [[f : τ₁ → τ₂, Γ]], (code, (clo, x₁, ..., xₙ))
               as ∃α((α × [[τ₁]]) → [[τ₂]]) × α)
            in clo
```

Problem?

Environment-passing closure conversion recursion

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

We can generalize the encoding afterwards,

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

whenever the right-hand side is well-defined.

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

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

Environment-passing closure conversion recursion

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

The encoding is simple.

However, this requires the introduction of recursive non-functional values "let rec x = v". While this is a useful construct, it really alters the operational semantics and requires updating the type soundness proof.

- Typed closure conversion

- Typed closure conversion
  - Environment passing
  - Closure passing

Closure-passing closure conversion

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

There are two difficulties:

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

Closure-passing closure conversion

There are two difficulties:

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

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

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

Tuples, rows, row variables

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

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

The notation $(\tau_1 \times \ldots \times \tau_n)$ was sugar for $\Pi\,(\tau_1; \ldots; \tau_n; \epsilon)$.

Let us now introduce *row variables* and allow *quantification* over them:

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

This allows reasoning about the first few fields of a tuple whose length is not known.

Typing rules for tuples

The typing rules for tuple construction and deconstruction are:

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

These rules make sense with or without row variables

Projection does not care about the fields beyond i. Thanks to row variables, this can be expressed in terms of *parametric polymorphism:*

$$\mathit{proj}_i : \forall \alpha._1 \ldots \alpha_i \rho.\, \Pi\,(\alpha_1; \ldots; \alpha_i; \rho) \to \alpha_i$$

About Rows

Rows were invented by Wand and improved by 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:

$[\![\tau_1 \to \tau_2]\!]$	
$= \quad \exists\rho.$	ρ describes the environment
$\quad \mu\alpha.$	α is the concrete type of the closure
$\quad\quad \Pi\,($	a tuple...
$\quad\quad\quad (\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!];$	...that begins with a code pointer...
$\quad\quad\quad \rho$	...and continues with the environment
$\quad\quad)$	

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

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

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

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

Closure-passing closure conversion

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

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

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

$$\begin{array}{rcl}
[\![\lambda x{:}[\![\tau_1]\!].\,M]\!] & = & \textit{let } code : (Clo([\![\Gamma]\!]) \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!] = \\
& & \quad \lambda(clo : Clo([\![\Gamma]\!], x : [\![\tau_1]\!]). \\
& & \qquad \textit{let } (\_, x_1, \ldots, x_n) : UClo[\![\Gamma]\!] = \textit{unfold } clo \textit{ in} \\
& & \qquad [\![M]\!] \textit{ in} \\
& & \textit{pack } [\![\Gamma]\!], (\textit{fold } (code, x_1, \ldots, x_n)) \\
& & \ \textit{as } \exists\rho.\,Clo(\rho)
\end{array}$$

$$\begin{array}{rcl}
[\![M_1\ M_2]\!] & = & \textit{let } \rho, clo = \textit{unpack } [\![M_1]\!] \textit{ in} \\
& & \textit{let } code : (Clo(\rho) \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!] = \\
& & \quad \textit{proj}_0\ (\textit{unfold } clo) \textit{ in} \\
& & code\ (clo, [\![M_2]\!])
\end{array}$$

Closure-passing closure conversion recursive functions

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

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

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

No extra field or extra work is required to store or construct a representation of the free variable f: the closure itself plays this role.

However, this untyped code can only be typechecked when recursion is monomorphic.

Exercise:

Check well-typedness with monomorphic recursion.

Closure-passing closure conversion recursive functions

The problem to adapt this encoding to polymorphic recursion is that recursive occurrences of f are rebuilt from the current invocation of the closure, *i.e.* is monomorphic since the closure is invoked after type specialization.

By contrast, in the environment passing encoding, the environment contained a polymorphic binding for the recursive calls that was filled with the closure before its invokation, *i.e.* with a polymorphic type.

Fortunately, we may slightly change the encoding, using a recursive closure as in the type-passing version, to allow typechecking in System F.

Closure-passing closure conversion recursive functions

Let τ be $\forall \vec{\alpha}.\, \tau_1 \to \tau_2$ and Γ_f be $f : \tau, \Gamma$ where $\vec{\beta} \# \Gamma$

$$
\begin{array}{l}
[\![\mu f\!:\!\tau.\, \lambda x.M]\!] = \textit{let } code = \\
\qquad \Lambda\vec{\beta}.\lambda(clo : Clo[\![\Gamma_f]\!], x : [\![\tau_1]\!]). \\
\qquad\quad \textit{let } (\_code, f, x_1, \ldots, x_n) : \forall\vec{\beta}.\, UClo([\![\Gamma_f]\!]) = \\
\qquad\qquad \textit{unfold } clo \textit{ in} \\
\qquad\quad [\![M]\!] \textit{ in} \\
\quad \textit{let rec } clo : \forall\vec{\beta}.\, \exists\rho.\, Clo(\rho) = \Lambda\vec{\beta}. \\
\qquad \textit{pack } [\![\Gamma]\!], (\textit{fold } (code\ \vec{\beta}, clo, x_1, \ldots, x_n)) \textit{ as } \exists\rho.\, Clo(\rho) \\
\quad \textit{in } clo
\end{array}
$$

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

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

Mutually recursive functions Environment passing

Can we compile mutually recursive functions?

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

Environment passing:

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

Mutually recursive functions Environment passing

Can we compile mutually recursive functions?

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

Environment passing:

```
[[M]]  =  let code_i = λ(env, x).
              let (f_1, f_2, x_1, ..., x_n) = env in
              [[M_i]]
          in
          let rec env = (clo_1, clo_2, x_1, ..., x_n)
              and clo_1 = (code_1, env)
              and clo_2 = (code_2, env) in
          clo_1, clo_2
```

Mutually recursive functions Environment passing

Can we compile mutually recursive functions?

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

Environment passing:

```
let $code_i$ = $\lambda(clo, x)$.
    let $(\_ , f_1, f_2, x_1, \ldots, x_n)$ = $clo$ in $[\![M_i]\!]$
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 Environment passing

Can we compile mutually recursive functions?

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

Environment passing:

```
let $code_1$ = $\lambda(clo, x)$.
   let $(\_code_1, \_code_2, f_1, f_2, x_1, \ldots, x_n)$ = $clo$ in $[\![M_1]\!]$ in
let $code_2$ = $\lambda(clo, x)$.
   let $(\_code_2, f_1, f_2, x_1, \ldots, x_n)$ = $clo$ in $[\![M_2]\!]$ in
let rec $clo_1$ = $(code_1, code_2, clo_1, clo_2, x_1, \ldots, x_n)$ and $clo_2$ = $c_1.tail$
 in $clo_1, clo_2$
```

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

Optimizing representations

Can closure passing and environment passing be mixed?

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

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

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

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

Encoding of objects

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

A class definition is an object generator:

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

Given arguments for parameter x_1, $\ldots x_1$, it will build recursive methods m_1, $\ldots m_n$.

Encoding of objects

A class can be compiled into an object closure:

$$\begin{array}{l} \textit{let } m = \\ \qquad \textit{let } m_1 = \lambda(m, x_1, \ldots, x_q).\, M_1 \textit{ in} \\ \qquad \ldots \\ \qquad \textit{let } m_p = \lambda(m, x_1, \ldots, x_q).\, M_p \textit{ in} \\ \qquad \{m_1, \ldots, m_p\} \;\; \textit{in} \\ \lambda x_1 \ldots x_q.\, (m, x_1, \ldots x_q) \end{array}$$

Each m_i is bound to the code for the corresponding method.
The code of all methods are combined into a record of methods, which is shared between all objects of the same class.

Calling method m_i of an object p is

$$(\textit{proj}_0\; p).m_i\; p$$

How can we type the encoding?

Typed encoding of objects

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

Let $Clo(R)$ be $\mu\alpha.\Pi(\{(m_i : \alpha \to \tau_i)^{i \in 1..n}\}; R)$ and $UClo(R)$ its unfolding.

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

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

Calling a method of an object p of type M is

$$p\#m_i \stackrel{\triangle}{=} \textit{let}\; \rho, z = \textit{unpack}\; p\; \textit{in}\; (\textit{proj}_0\; \textit{unfold}\; z).m_i\; z$$

An object has a recursive type but it is *not* a recursive value.

Typed encoding of objects

Typed encoding of objects were first studied in the 90's to understand what objects really are in a type setting.

These encodings are in fact type-preserving compilation of (primitive) objects.

There are several variations on these encodings. See [Bruce et al., 1999] for a comparison.

See [Rémy, 1994] for an encoding of objects in (a small extension of) ML with iso-existentials and universals.

See [Abadi and Cardelli, 1996, 1995] for more details on primitive objects.

Moral of the story

Type-preserving compilation is rather *fun.* (Yes, really!)

It forces compiler writers to make the structure of the compiled program *fully explicit,* in type-theoretic terms.

In practice, building explicit type derivations, ensuring that they remain small and can be efficiently typechecked, can be a lot of work.

Optimizations

Because we have focused on type preservation, we have studied only 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].