MPRI, Typage

Didier Rémy
(With much course meterial from François Pottier)

October 5, 2010

Plan of the course

Introduction

Simply-typed λ-calculus

Polymorphism and System F

Type reconstruction

Existential types

Existential types

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 language with usually richer type systems.

The encoding may be sometimes be used directly as a programming idioms in the source language. For Example:

- Closure conversion requires an extension of the language with existential types, which happens to very useful on their own.
- Closures themselves are themselves a simple form of objects.
- Defunctionalization may be done manually on some particular program, *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 t to $[\![t]\!]$, but also a translation of *types*, mapping T to $[\![T]\!]$, with the property:

$$\Gamma \vdash t : T \quad \text{implies} \quad [\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]$$

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

Contents

- 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 contains 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 term translations is minor.

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

Closure-passing closure conversion

The closure-passing variant is as follows:

$$
\begin{aligned}
[\![\lambda x.\, t]\!] \quad = \quad & \text{let } code = \lambda(clo, x). \\
& \quad \text{let } (\_, x_1, \ldots, x_n) = clo \text{ in} \\
& \quad [\![t]\!] \text{ in} \\
& (code, x_1, \ldots, x_n)
\end{aligned}
$$

$$
\begin{aligned}
[\![t_1\ t_2]\!] \quad = \quad & \text{let } clo = [\![t_1]\!] \text{ in} \\
& \text{let } code = \text{proj}_0\ clo \text{ in} \\
& code\ (clo, [\![t_2]\!])
\end{aligned}
$$

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

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

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

Environment-passing closure conversion

The environment-passing variant is as follows:

$$
\begin{aligned}
[\![\lambda x.\, t]\!] \quad = \quad & \mathsf{let}\ code = \lambda(env, x). \\
& \quad \mathsf{let}\ (x_1, \ldots, x_n) = env\ \mathsf{in} \\
& \quad [\![t]\!]\ \mathsf{in} \\
& (code, (x_1, \ldots, x_n))
\end{aligned}
$$

$$
\begin{aligned}
[\![t_1\ t_2]\!] \quad = \quad & \mathsf{let}\ (code, env) = [\![t_1]\!]\ \mathsf{in} \\
& \\
& code\ (env, [\![t_2]\!])
\end{aligned}
$$

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

Towards type-preserving closure conversion

Let us first focus on the environment-passing variant.

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, $[\![T_1 \to T_2]\!]$?

Towards type-preserving closure conversion

Let us examine the closure allocation code again:

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

Suppose $\Gamma \vdash \lambda x.\, t : T_1 \to T_2$.

Suppose, without loss of generality, $dom(\Gamma) = fv(\lambda x.\, t) = \{x_1, \ldots, x_n\}$.

Overloading notation, if Γ is $x_1 : T_1; \ldots; x_n : T_n$, write $[\![\Gamma]\!]$ for the tuple type $T_1 \times \ldots \times T_n$.

By hypothesis, we have $[\![\Gamma]\!], x : [\![T_1]\!] \vdash [\![t]\!] : [\![T_2]\!]$, so env has type $[\![\Gamma]\!]$, $code$ has type $([\![\Gamma]\!] \times [\![T_1]\!]) \to [\![T_2]\!]$, and the entire closure has type $(([\![\Gamma]\!] \times [\![T_1]\!]) \to [\![T_2]\!]) \times [\![\Gamma]\!]$.

Now, *what should be the definition of* $[\![T_1 \to T_2]\!]$*?*

A weakening rule

(Parenthesis.)

In order to support the hypothesis $dom(\Gamma) = fv(\lambda x.t)$ at every λ-abstraction, it is possible to introduce an (admissible) *weakening* rule:

$$\text{Weakening}\quad \frac{\Gamma_1;\Gamma_2 \vdash t : T \qquad x \# t}{\Gamma_1;x : T';\Gamma_2 \vdash t : T}$$

If the weakening rule is applied eagerly at every λ-abstraction, then the hypothesis is met, and closures have *minimal environments.*

(In some cases, one may not use minimal environments, *e.g.* to allow sharing of environments between several closures.)

Towards a type translation

Can we adopt this as a definition?

$$[\![T_1 \to T_2]\!] \quad = \quad (([\![\Gamma]\!] \times [\![T_1]\!]) \to [\![T_2]\!]) \times [\![\Gamma]\!]$$

Towards a type translation

Can we adopt this as a definition?

$$[\![T_1 \to T_2]\!] \quad = \quad ((\Gamma \times [\![T_1]\!]) \to [\![T_2]\!]) \times \Gamma$$

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:

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

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* $[\![T_1 \to T_2]\!]$*?*

The type translation

The only sensible solution is:

$$[\![T_1 \to T_2]\!] \quad = \quad \exists X.((X \times [\![T_1]\!]) \to [\![T_2]\!]) \times X$$

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 X 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

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

Existential types

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

$$T ::= \ldots \mid \exists X.T$$

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

$$\frac{\Gamma \vdash t : [X \mapsto T']T}{\Gamma \vdash \mathsf{pack}\ T', t\ \mathsf{as}\ \exists X.T : \exists X.T}$$

Unpack

$$\frac{\Gamma \vdash t_1 : \exists X.T_1 \qquad \Gamma, X, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \mathsf{let}\ X, x = \mathsf{unpack}\ t_1\ \mathsf{in}\ t_2 : T_2}$$

Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

$$\text{Pack}\quad \frac{\Gamma \vdash t : [X \mapsto T']T}{\Gamma \vdash \text{pack } T', t \text{ as } \exists X.T : \exists X.T} \qquad \text{Unpack}\quad \frac{\Gamma \vdash t_1 : \exists X.T_1 \quad \Gamma, X, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2}$$

Anything wrong?

Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

$$\text{Pack}\quad \frac{\Gamma \vdash t : [X \mapsto T']T}{\Gamma \vdash \mathsf{pack}\, T', t\ \mathsf{as}\ \exists X.T : \exists X.T} \qquad \text{Unpack}\quad \frac{\Gamma \vdash t_1 : \exists X.T_1 \qquad \Gamma, X, x : T_1 \vdash t_2 : T_2 \qquad X \# T_2}{\Gamma \vdash \mathsf{let}\ X, x = \mathsf{unpack}\ t_1\ \mathsf{in}\ t_2 : T_2}$$

The side condition $X \# T_2$ is mandatory here to ensure well-formedness of the conclusion. If well-formedness conditions were explicit in judgments, then $\Gamma \vdash T_2$ would suffice, as it would imply $X \# T_2$, since the last premise implies $X \# \Gamma$.

Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

$$\text{Pack}\quad \frac{\Gamma \vdash t : [X \mapsto T']T}{\Gamma \vdash \mathsf{pack}\ T', t\ \mathsf{as}\ \exists X.T : \exists X.T} \qquad \text{Unpack}\quad \frac{\Gamma \vdash t_1 : \exists X.T_1 \quad \Gamma, X, x : T_1 \vdash t_2 : T_2 \quad X \# T_2}{\Gamma \vdash \mathsf{let}\ X, x = \mathsf{unpack}\ t_1\ \mathsf{in}\ t_2 : T_2}$$

The side condition $X \# T_2$ is mandatory here to ensure well-formedness of the conclusion. If well-formedness conditions were explicit in judgments, then $\Gamma \vdash T_2$ would suffice, as it would imply $X \# T_2$, since the last premise implies $X \# \Gamma$.

Note the *imperfect duality* between universals and existentials:

$$\text{TAbs}\quad \frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T} \qquad \text{TApp}\quad \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t\,T' : [X \mapsto T']T}$$

On existential elimination

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

$$\frac{\Gamma, X \vdash t : \exists X.T}{\Gamma, X \vdash \mathsf{unpack}\, t : T}$$

Informally, this could mean that, it t has type T for some *unknown* X, then it has type T, where X is "fresh"...

Why is this broken?

On existential elimination

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

$$\frac{\Gamma, X \vdash t : \exists X.T}{\Gamma, X \vdash \mathsf{unpack}\, t : T}$$

Informally, this could mean that, it t has type T for some *unknown* X, then it has type T, where X is "fresh"...

Why is this broken?

We can immediately *universally* quantify over X, and conclude that $\Gamma \vdash t : \forall X.T$. This is nonsense!

Replacing the premise $\Gamma, X \vdash t : \exists X.T$ by the conjunction $\Gamma \vdash t : \exists X.T$ and $X \in \mathit{dom}(\Gamma)$ 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 X.

Hence, the elimination rule must have control over the *user* of the package — that is, over the term t_2.

Unpack

$$\frac{\Gamma \vdash t_1 : \exists X.T_1 \qquad \Gamma, X; x : T_1 \vdash t_2 : T_2 \qquad X \# T_2}{\Gamma \vdash \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2}$$

The restriction $X \# T_2$ prevents writing "let $X, x = \text{unpack } t_1$ in x", which would be equivalent to the unsound "unpack t" of the previous slide.

The fact that X is bound within t_2 forces it to be treated abstractly.

In fact, t_2 must be ... in the type-variable X.

On existential elimination

In fact, t_2 must be *polymorphic* in X. The rule could be written:

$$\frac{\Gamma \vdash t_1 : \exists X.T_1 \qquad \Gamma \vdash \Lambda X.\lambda x.\, t_2 : \forall X.\, T_1 \to T_2 \qquad X \# T_2}{\Gamma \vdash \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2}$$

or, more economically:

$$\frac{\Gamma \vdash t_1 : \exists X.T_1 \qquad \Gamma \vdash t_2 : \forall X.\, T_1 \to T_2 \qquad X \# T_2}{\Gamma \vdash \text{unpack } t_1\ t_2 : T_2}$$

On existential elimination

In fact, t_2 must be *polymorphic* in X. The rule could be written:

$$\frac{\Gamma \vdash t_1 : \exists X.T_1 \qquad \Gamma \vdash \Lambda X.\lambda x.\, t_2 : \forall X.\, T_1 \to T_2 \qquad X \# T_2}{\Gamma \vdash \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2}$$

or, more economically:

$$\frac{\Gamma \vdash t_1 : \exists X.T_1 \qquad \Gamma \vdash t_2 : \forall X.\, T_1 \to T_2 \qquad X \# T_2}{\Gamma \vdash \text{unpack } t_1\ t_2 : T_2}$$

One could even view "$\text{unpack}_{\exists X.T}$" as a *constant*, equipped with an appropriate type:

On existential elimination

In fact, t_2 must be *polymorphic* in X. The rule could be written:

$$\frac{\Gamma \vdash t_1 : \exists X.T_1 \qquad \Gamma \vdash \Lambda X.\lambda x.\, t_2 : \forall X.\, T_1 \to T_2 \qquad X \# T_2}{\Gamma \vdash \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2}$$

or, more economically:

$$\frac{\Gamma \vdash t_1 : \exists X.T_1 \qquad \Gamma \vdash t_2 : \forall X.\, T_1 \to T_2 \qquad X \# T_2}{\Gamma \vdash \text{unpack } t_1\ t_2 : T_2}$$

One could even view "$\text{unpack}_{\exists X.T}$" as a *constant*, equipped with an appropriate type:

$$\text{unpack}_{\exists X.T} : \quad \exists X.T \to \forall Y.\, ((\forall X.\, (T \to Y)) \to Y)$$

The variable Y, which stands for T_2, is bound prior to X, so it naturally cannot be instantiated to a type that refers to X. This reflects the side condition $X \# T_2$.

On existential introduction

$$\text{Pack}\ \frac{\Gamma \vdash t : [X \mapsto T']T}{\Gamma \vdash \mathsf{pack}\ T', t\ \mathsf{as}\ \exists X.T : \exists X.T}$$

If *desired*, "$\mathsf{pack}_{\exists X.T}$" *could also be viewed as a constant*:

$$\mathsf{pack}_{\exists X.T} : \quad \forall X.(T \to \exists X.T)$$

Existentials as constants

In summary, System F with existential types can also be presented as follows:

$$\begin{aligned} \mathrm{pack}_{\exists X.T} &: \forall X.\,(T \to \exists X.T) \\ \mathrm{unpack}_{\exists X.T} &: \exists X.\,T \to \forall Y.\,((\forall X.\,(T \to Y)) \to Y) \end{aligned}$$

These can be read as follows:

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

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

Is there an encoding of existential types into universal types?

Encoding existentials into universals

The type translation is *double negation:*

$$[\![\exists X.T]\!] \;=\; \forall Y.((\forall X.([\![T]\!] \to Y)) \to Y) \qquad \text{if } Y \# T$$

The term translation is:

$$\begin{array}{rcl} [\![\mathsf{pack}_{\exists X.T}]\!] & : & \forall X.([\![T]\!] \to [\![\exists X.T]\!]) \\ & = & ? \end{array}$$

$$\begin{array}{rcl} [\![\mathsf{unpack}_{\exists X.T}]\!] & : & [\![\exists X.T]\!] \to \forall Y.((\forall X.([\![T]\!] \to Y)) \to Y) \\ & = & ? \end{array}$$

Encoding existentials into universals

The type translation is *double negation:*

$$[\![\exists X.T]\!] \;=\; \forall Y.((\forall X.([\![T]\!] \to Y)) \to Y) \qquad \text{if } Y \# T$$

The term translation is:

$$\begin{aligned} [\![\mathsf{pack}_{\exists X.T}]\!] \;&:\; \forall X.([\![T]\!] \to [\![\exists X.T]\!]) \\ &=\; \Lambda X.\lambda x{:}[\![T]\!].\Lambda Y.\lambda k{:}\forall X.([\![T]\!] \to Y).\, ? : Y \\ [\![\mathsf{unpack}_{\exists X.T}]\!] \;&:\; [\![\exists X.T]\!] \to \forall Y.((\forall X.([\![T]\!] \to Y)) \to Y) \\ &=\; ? \end{aligned}$$

Encoding existentials into universals

The type translation is *double negation:*

$$[\![\exists X.T]\!] \;=\; \forall Y.((\forall X.([\![T]\!] \to Y)) \to Y) \qquad \text{if } Y \# T$$

The term translation is:

$$\begin{array}{rcl} [\![\mathsf{pack}_{\exists X.T}]\!] & : & \forall X.([\![T]\!] \to [\![\exists X.T]\!]) \\ & = & \Lambda X.\lambda x{:}[\![T]\!].\,\Lambda Y.\lambda k{:}\forall X.([\![T]\!] \to Y).\,k\ X\ x \\ [\![\mathsf{unpack}_{\exists X.T}]\!] & : & [\![\exists X.T]\!] \to \forall Y.((\forall X.([\![T]\!] \to Y)) \to Y) \\ & = & ? \end{array}$$

Encoding existentials into universals

The type translation is *double negation:*

$$[\![\exists X.T]\!] \;=\; \forall Y.((\forall X.([\![T]\!] \to Y)) \to Y) \qquad \text{if } Y \# T$$

The term translation is:

$$\begin{aligned} [\![\mathsf{pack}_{\exists X.T}]\!] \;&:\; \forall X.([\![T]\!] \to [\![\exists X.T]\!]) \\ &=\; \Lambda X.\lambda x{:}[\![T]\!].\,\Lambda Y.\lambda k{:}\forall X.([\![T]\!] \to Y).\,k\ X\ x \end{aligned}$$

$$\begin{aligned} [\![\mathsf{unpack}_{\exists X.T}]\!] \;&:\; [\![\exists X.T]\!] \to \forall Y.((\forall X.([\![T]\!] \to Y)) \to Y) \\ &=\; \lambda x{:}[\![\exists X.T]\!].\,x \end{aligned}$$

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

What is the computational content of this encoding?

Encoding existentials into universals

The type translation is *double negation:*

$$[\![\exists X.T]\!] \;=\; \forall Y.((\forall X.([\![T]\!] \to Y)) \to Y) \qquad \text{if } Y \# T$$

The term translation is:

$$\begin{aligned} [\![\mathsf{pack}_{\exists X.T}]\!] \;&:\; \forall X.([\![T]\!] \to [\![\exists X.T]\!]) \\ &=\; \Lambda X.\lambda x{:}[\![T]\!].\Lambda Y.\lambda k{:}\forall X.([\![T]\!] \to Y).k\ X\ x \\ [\![\mathsf{unpack}_{\exists X.T}]\!] \;&:\; [\![\exists X.T]\!] \to \forall Y.((\forall X.([\![T]\!] \to Y)) \to Y) \\ &=\; \lambda x{:}[\![\exists X.T]\!].x \end{aligned}$$

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

What is the computational content of this encoding?

A *continuation-passing transform.*

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

The semantics of existential types as constants

$\mathsf{pack}_{\exists X.T}$ and $\mathsf{unpack}_{\exists X.T}$ can be treated as a unary constructor and as a binary destructor, respectively, with the following reduction rule:

$$\mathsf{unpack}_{\exists X.T}\ (\mathsf{pack}_{\exists X.T}, T'\ v)\ \ (\Lambda Y.\lambda y{:}\forall X.T \to Y.t) \longrightarrow t\ T'\ v$$

Exercise

Show that they satisfy the progress and subject reduction assumptions for constants.

The semantics of existential types as primitive

We extend values and evaluation contexts as follows:

$$\begin{array}{rcl} v & ::= & \ldots \mid \mathsf{pack}\ T', v\ \mathsf{as}\ T \\ E & ::= & \ldots \mid \mathsf{pack}\ T', E\ \mathsf{as}\ T \mid \mathsf{let}\ X, x = \mathsf{unpack}\ E\ \mathsf{in}\ t \end{array}$$

We add the reduction rule:

$$\mathsf{let}\ X, x = \mathsf{unpack}\ (\mathsf{pack}\ T', v\ \mathsf{as}\ T)\ \mathsf{in}\ t \longrightarrow [X \mapsto T'][x \mapsto v]t$$

Exercise

Show that subject reduction and progress holds.

The semantics of existential types beware!

The reduction rule for existential destructs its arguments. Hence, let $X, x = \text{unpack } t_1 \text{ in } t_2$ cannot be reduced unless t_1 is itself a packed expression, which is indeed the case when t_1 is value (or in head normal form).

This contrasts with $\text{let } x : T = t_1 \text{ in } t_2$ where t_1 need not be evaluation and may be an application (*e.g.* in call by name or with strong reduction).

The semantics of existential types beware!

Exercise

Find an example that illustrates why the reduction of let $X, x = \text{unpack } t_1 \text{ in } t_2$ *could be problematic when* t_1 *is not a value.*

The semantics of existential types beware!

Exercise

Find an example that illustrates why the reduction of let $X, x = \text{unpack } t_1 \text{ in } t_2$ *could be problematic when* t_1 *is not a value.*

Need a hint?

Use a conditional

The semantics of existential types beware!

Exercise

Find an example that illustrates why the reduction of let $X, x = \text{unpack } t_1 \text{ in } t_2$ *could be problematic when* t_1 *is not a value.*

Solution

Let t_1 be if t then v_1 else v_2 where v_i is of the form pack T_i, v_i as $\exists XT$ and the two witnesses T_1 and T_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 t_1, before unpacking.

Is pack too verbose?

Exercise

Recall the typing rule for pack:

$$\frac{\Gamma \vdash t : [X \mapsto T']T}{\Gamma \vdash \mathsf{pack}\ T', t\ \mathsf{as}\ \exists X.T : \exists X.T}$$

Isn't the witness type T' annotation superfluous?

Is pack too verbose?

Exercise

Recall the typing rule for pack:

$$\frac{\Gamma \vdash t : [X \mapsto T']T}{\Gamma \vdash \text{pack } T', t \text{ as } \exists X.T : \exists X.T}$$

Isn't the witness type T' annotation superfluous?

- The type T_0 of t is fully determined by t and the given type $\exists X.T$ of the packed value. Checking that T_0 is of the form $[X \mapsto T']T$ t is the matching problem for second-order types, which is simple.
- However, the reduction rule need the witness type T'. 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.

- 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 information and can just be dropped in the syntax of terms.

There just remains a let-binding form instead of the unpack form.

After type-erasure, the typing rules are:

Unpack

$$\frac{\Gamma \vdash a_1 : \exists X.T_1 \qquad \Gamma, X, x : T_1 \vdash a_2 : T_2 \qquad X \# T_2}{\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : T_2}$$

Pack

$$\frac{\Gamma \vdash a : [X \mapsto T']T}{\Gamma \vdash a : \exists X.T}$$

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

The semantics of the let-binding as before:

$$E ::= \ldots \mid \mathsf{let}\ x = E\ \mathsf{in}\ t \qquad\qquad \mathsf{let}\ x = v\ \mathsf{in}\ t \longrightarrow [x \mapsto v]t$$

Is the semantics type-erasing?

Implicitly-typed existential types

subtlety

Yes, it is.

But this is a subtlety!

Implicitly-typed existential types

subtlety

Yes, it is.

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

Implicitly-typed existential types

subtlety

Yes, it is.

But this 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 call-by-name evaluation the arguments of let-bindings are not reduced prior to substitution of the argument:

$$\text{let } x = t_1 \text{ in } t_2 \longrightarrow [x \mapsto t_1]t_2$$

With existential types, this breaks subject reduction!

Why?

Implicitly-typed existential types

subtlety

Let T_0 be $\exists X.\,(X \to X) \to (X \to X)$, w_0 a value of type bool, w_1 and w_2 two values of type T_0 with incompatible witness types, for instance, $\lambda f.\,\lambda x.\,1 + (f\ (1 + x))$ and $\lambda f.\,\lambda x.\,not\ (f\ (not\ x))$. Let w be the function $\lambda b.\,\text{if } b \text{ then } w_1 \text{ else } w_2$ of type $bool \to T_0$.

$$a_1 \;=\; \text{let } x = w\ w_0 \text{ in } x\ (x\ (\lambda y.\,y)) \;\longrightarrow\; w\ w_0\ (w\ w_0\ (\lambda y.\,y)) \;=\; a_2$$

We have $\varnothing \vdash a_1 : \exists X.\,X \to X$ while $\varnothing \not\vdash a_2 : T$.

What happened?

Implicitly-typed existential types — subtlety

Let T_0 be $\exists X.\,(X \to X) \to (X \to X)$, w_0 a value of type bool, w_1 and w_2 two values of type T_0 with incompatible witness types, for instance, $\lambda f.\,\lambda x.\,1 + (f\ (1 + x))$ and $\lambda f.\,\lambda x.\,not\ (f\ (not\ x))$. Let w be the function $\lambda b.\,\text{if } b \text{ then } w_1 \text{ else } w_2$ of type $\text{bool} \to T_0$.

$$a_1 \;=\; \text{let } x = w\,w_0 \text{ in } x\ (x\ (\lambda y.\,y)) \;\longrightarrow\; w\,w_0\ (w\,w_0\ (\lambda y.\,y)) \;=\; a_2$$

We have $\varnothing \vdash a_1 : \exists X.\,X \to X$ while $\varnothing \not\vdash a_2 : T$.

The term a_1 is well-typed since $w\,w_0$ has type T_0, hence x can be assumed of type $(Y \to Y) \to (Y \to Y)$ for some unknown type Y and $\lambda y.\,y$ is of type $Y \to Y$.

However, without the outer existential type $w\,w_0$ can only be typed with $(\forall X.\,X \to X) \to \exists X.\,(X \to X)$, because the value returned by the function need different witnesses for X. This is requiring too much on its argument and the outer application is ill-typed.

Implicitly-typed existential types

subtlety

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

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

$$\frac{\Gamma \vdash a_1 : \exists X.T_1 \qquad \Gamma, X, x : T_1 \vdash a_2 : T_2 \qquad X \# T_2}{\Gamma \vdash [x \mapsto a_1]a_2 : T_2}$$

Comments?

Implicitly-typed existential types

subtlety

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

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

$$\frac{\Gamma \vdash a_1 : \exists X.T_1 \qquad \Gamma, X, x : T_1 \vdash a_2 : T_2 \qquad X \# T_2}{\Gamma \vdash [x \mapsto a_1]a_2 : T_2}$$

Comments?

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

Implicitly-typed existential types

subtlety

Existential types are more tricky 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 be recovered by further reduction steps.

Implicitly-typed existential types *encoding*

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

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

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

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

- 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:*

$$D\,\vec{X} \approx \exists \bar{Y}.T \qquad \text{if } \mathrm{ftv}(T) \subseteq \bar{X} \cup \bar{Y} \quad \text{and} \quad \bar{X} \# \bar{Y}$$

This introduces two constants, with the following type schemes:

$$\begin{aligned} \mathrm{pack}_D &: \forall \bar{X}\bar{Y}.T \to D\,\vec{X} \\ \mathrm{unpack}_D &: \forall \bar{X}Z.D\,\vec{X} \to (\forall \bar{Y}.(T \to Z)) \to Z \end{aligned}$$

(Compare with basic iso-recursive types, where $\bar{Y} = \emptyset$.)

Iso-existential types in ML

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

$$\forall \bar{X} Z.\, D\, \vec{X} \to (\forall \bar{Y}.\, (T \to Z)) \to Z$$

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:"

$$\forall \bar{X} Z. D\,\vec{X} \to (\forall \bar{Y}. (T \to Z)) \to Z$$

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

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

Unpack_D

$$\frac{\Gamma \vdash t_1 : D\,\vec{T} \qquad \Gamma \vdash t_2 : \forall \bar{Y}. ([\vec{X} \mapsto \vec{T}]T \to T_2) \qquad \bar{Y} \# \vec{T}, T_2}{\Gamma \vdash \text{unpack}_D\ t_1\ t_2 : T_2} \quad \text{where } D\,\vec{X} \approx \exists \bar{Y}. T$$

We have seen a version of this rule in System F earlier; this in an ML version. The term t_2 must be polymorphic, which *Gen can prove.*

Iso-existential types in ML

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

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

The construct unpack_D leads to this constraint generation rule:

$$[\![\text{unpack}_D\ t_1\ t_2 : T_2]\!] \quad = \quad \exists \bar{X}. \left(\begin{array}{l} [\![t_1 : D\ \vec{X}]\!] \\ \forall \bar{Y}. [\![t_2 : T \to T_2]\!] \end{array} \right)$$

where $D\ \vec{X} \approx \exists \bar{Y}.T$ and, *w.l.o.g.*, $\bar{X}\bar{Y} \# t_1, t_2, T_2$.

Again, 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.

The (somewhat bizarre) Haskell syntax for this is:

$$\text{data } D\ \vec{X} = \text{forall } \bar{Y}.\ell\ T$$

where ℓ is a data constructor. The elimination construct becomes:

$$[\![\text{case } t_1 \text{ of } \ell\ x \to t_2 : T_2]\!] \quad = \quad \exists \bar{X}. \left(\begin{array}{l} [\![t_1 : D\ \vec{X}]\!] \\ \forall \bar{Y}.\, \text{def } x : T \text{ in } [\![t_2 : T_2]\!] \end{array} \right)$$

where, w.l.o.g., $\bar{X}\bar{Y} \# t_1, t_2, T_2$.

An example

Define $\mathsf{Any} \approx \exists Y.Y$. An attempt to extract the raw contents of a package fails:

$$\begin{aligned}
[\![\mathsf{unpack}_{\mathsf{Any}}\ t_1\ (\lambda x.x) : T_2]\!] &= [\![t_1 : \mathsf{Any}]\!] \wedge \forall Y.[\![\lambda x.x : Y \to T_2]\!] \\
&\Vdash \forall Y.Y = T_2 \\
&\equiv \mathsf{false}
\end{aligned}$$

(Recall that $Y \# T_2$.)

An example

Define

$$D\ X \approx \exists Y.(Y \to X) \times Y$$

A client that regards Y as abstract succeeds:

$$\begin{aligned}
& [\![\mathsf{unpack}_D\ t_1\ (\lambda(f, y).f\ y) : T]\!] \\
= \quad & \exists X.([\![t_1 : D\ X]\!] \wedge \forall Y.[\![\lambda(f, y).f\ y : ((Y \to X) \times Y) \to T]\!]) \\
\equiv \quad & \exists X.([\![t_1 : D\ X]\!] \wedge \forall Y.\mathsf{def}\ f : Y \to X; y : Y\ \mathsf{in}\ [\![f\ y : T]\!]) \\
\equiv \quad & \exists X.([\![t_1 : D\ X]\!] \wedge \forall Y.T = X) \\
\equiv \quad & \exists X.([\![t_1 : D\ X]\!] \wedge T = X) \\
\equiv \quad & [\![t_1 : D\ T]\!]
\end{aligned}$$

Existential types calls for universal types!

Exercise Reusing the type $D\,X \approx \exists Y.(Y \to X) \times Y$ of frozen computations, assume given a list l of whose of elements of type $D\,T_1$.

Assume given a function g of type $T_1 \to T_2$. Transforms the list into a new list l' of frozen computations of type $D\,T_2$ (without actually running any computation).

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

Generalize into a functional that receives g and into a function that receives g and l and returns l'.

Unfortunately, the following code does not typecheck:

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

In expression let $X, x =$ unpack t_1 in t_2, occurrences of x can only be passed to polymorphic functions so that x does not leek 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:

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

specifies an abstract implementation of integer stacks.

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

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

Contents

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

- 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:

$$\Gamma \vdash t : T \quad \text{implies} \quad [\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]$$

Environment-passing closure conversion

Assume $\Gamma \vdash \lambda x.t : T_1 \to T_2$ and $dom(\Gamma) = \{x_1, \ldots, x_n\} = fv(\lambda x.t)$.

$$
\begin{array}{rcl}
[\![\lambda x{:}T_1.t]\!] & = & \text{let } code : \qquad\qquad\qquad\qquad = \\
& & \quad \lambda(env : \quad , x : \quad). \\
& & \quad\quad \text{let } (x_1, \ldots, x_n : \quad) = env \text{ in} \\
& & \quad\quad [\![t]\!] \\
& & \text{in} \\
& & \text{pack} \quad , (code, (x_1, \ldots, x_n)) \\
& & \text{as}
\end{array}
$$

We find $[\![\Gamma]\!] \vdash [\![\lambda x{:}T_1.t]\!] : [\![T_1 \to T_2]\!]$, as desired.

Environment-passing closure conversion

Assume $\Gamma \vdash \lambda x.t : T_1 \to T_2$ and $dom(\Gamma) = \{x_1, \ldots, x_n\} = fv(\lambda x.t)$.

$$
\begin{array}{rcl}
[\![\lambda x{:}T_1.t]\!] & = & \mathsf{let}\ code : \qquad\qquad\qquad\qquad\qquad\qquad = \\
& & \quad \lambda(env : [\![\Gamma]\!], x : [\![T_1]\!]). \\
& & \quad\quad \mathsf{let}\ (x_1, \ldots, x_n : [\![\Gamma]\!]) = env\ \mathsf{in} \\
& & \quad\quad [\![t]\!] \\
& & \mathsf{in} \\
& & \mathsf{pack}\quad , (code, (x_1, \ldots, x_n)) \\
& & \mathsf{as}
\end{array}
$$

We find $[\![\Gamma]\!] \vdash [\![\lambda x{:}T_1.t]\!] : [\![T_1 \to T_2]\!]$, as desired.

Environment-passing closure conversion

Assume $\Gamma \vdash \lambda x.t : T_1 \to T_2$ and $dom(\Gamma) = \{x_1, \ldots, x_n\} = fv(\lambda x.t)$.

$$
\begin{aligned}
[\![\lambda x{:}T_1.t]\!] \quad = \quad & \text{let } code : ([\![\Gamma]\!] \times [\![T_1]\!]) \to [\![T_2]\!] = \\
& \quad \lambda(env : [\![\Gamma]\!], x : [\![T_1]\!]). \\
& \quad\quad \text{let } (x_1, \ldots, x_n : [\![\Gamma]\!]) = env \text{ in} \\
& \quad\quad [\![t]\!] \\
& \text{in} \\
& \text{pack} \quad , (code, (x_1, \ldots, x_n)) \\
& \text{as}
\end{aligned}
$$

We find $[\![\Gamma]\!] \vdash [\![\lambda x{:}T_1.t]\!] : [\![T_1 \to T_2]\!]$, as desired.

Environment-passing closure conversion

Assume $\Gamma \vdash \lambda x.t : T_1 \to T_2$ and $dom(\Gamma) = \{x_1, \ldots, x_n\} = fv(\lambda x.t)$.

$$
\begin{aligned}
[\![\lambda x{:}T_1.t]\!] \quad = \quad & \text{let } code : ([\![\Gamma]\!] \times [\![T_1]\!]) \to [\![T_2]\!] = \\
& \quad \lambda(env : [\![\Gamma]\!], x : [\![T_1]\!]). \\
& \quad\quad \text{let } (x_1, \ldots, x_n : [\![\Gamma]\!]) = env \text{ in} \\
& \quad\quad [\![t]\!] \\
& \text{in} \\
& \text{pack } [\![\Gamma]\!], (code, (x_1, \ldots, x_n)) \\
& \text{as } \exists X((X \times [\![T_1]\!]) \to [\![T_2]\!]) \times X
\end{aligned}
$$

We find $[\![\Gamma]\!] \vdash [\![\lambda x{:}T_1.t]\!] : [\![T_1 \to T_2]\!]$, as desired.

Environment-passing closure conversion

Assume $\Gamma \vdash t : T_1 \to T_2$ and $\Gamma \vdash t_1 : T_1$.

$$\begin{aligned}[\![t\ t_1]\!] \quad = \quad & \text{let } X, (\mathit{code} : (X \times [\![T_1]\!]) \to T_2, \mathit{env} : X) = \\ & \quad \text{unpack } [\![t]\!] \text{ in} \\ & \quad \mathit{code}\ (\mathit{env}, [\![t_1]\!])\end{aligned}$$

We find $[\![\Gamma]\!] \vdash [\![t\ t_1]\!] : [\![T_2]\!]$, as desired.

Environment-passing closure conversion recursion

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

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

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

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

What is the weak point of this variant?

Environment-passing closure conversion recursion

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

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

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

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:

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

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

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

Environment-passing closure conversion

Here is how the "fix-pack" variant is type-checked. Assume $\Gamma \vdash \mu f.\lambda x.t : T_1 \rightarrow T_2$ and $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\mu f.\lambda x.t)$.

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

Problem?

Environment-passing closure conversion

Here is how the "fix-pack" variant is type-checked. Assume $\Gamma \vdash \mu f.\lambda x.t : T_1 \rightarrow T_2$ and $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\mu f.\lambda x.t)$.

$$
\begin{array}{l}
[\![\mu f : T_1 \rightarrow T_2.\lambda x.t]\!] = \\
\qquad \text{let } code : \qquad\qquad\qquad\qquad = \\
\qquad\quad \lambda(env : [\![f : T_1 \rightarrow T_2, \Gamma]\!], x : [\![T_1]\!]). \\
\qquad\qquad \text{let } (f, x_1, \ldots, x_n) : [\![f : T_1 \rightarrow T_2, \Gamma]\!] = env \text{ in} \\
\qquad\qquad [\![t]\!] \text{ in} \\
\qquad \text{let rec } clo : \qquad\qquad = \\
\qquad\quad \text{pack } [\![f : T_1 \rightarrow T_2, \Gamma]\!], (code, (clo, x_1, \ldots, x_n)) \\
\qquad\quad \text{as} \\
\qquad \text{in } clo
\end{array}
$$

Problem?

Environment-passing closure conversion

Here is how the "fix-pack" variant is type-checked. Assume $\Gamma \vdash \mu f.\lambda x.t : T_1 \to T_2$ and $dom(\Gamma) = \{x_1, \ldots, x_n\} = fv(\mu f.\lambda x.t)$.

$$
\begin{array}{l}
[\![\mu f : T_1 \to T_2.\lambda x.t]\!] = \\
\quad \text{let } code : ([\![f : T_1 \to T_2; \Gamma]\!] \times [\![T_1]\!]) \to [\![T_2]\!] = \\
\quad\quad \lambda(env : [\![f : T_1 \to T_2, \Gamma]\!], x : [\![T_1]\!]). \\
\quad\quad\quad \text{let } (f, x_1, \ldots, x_n) : [\![f : T_1 \to T_2, \Gamma]\!] = env \text{ in} \\
\quad\quad\quad [\![t]\!] \text{ in} \\
\quad \text{let rec } clo : \qquad\qquad = \\
\quad\quad \text{pack } [\![f : T_1 \to T_2, \Gamma]\!], (code, (clo, x_1, \ldots, x_n)) \\
\quad\quad as \\
\quad \text{in } clo
\end{array}
$$

Problem?

Environment-passing closure conversion

Here is how the "fix-pack" variant is type-checked. Assume $\Gamma \vdash \mu f.\lambda x.t : T_1 \to T_2$ and $dom(\Gamma) = \{x_1, \ldots, x_n\} = fv(\mu f.\lambda x.t)$.

$$
\begin{array}{l}
[\![\mu f : T_1 \to T_2.\lambda x.t]\!] = \\
\quad \text{let } code : ([\![f : T_1 \to T_2; \Gamma]\!] \times [\![T_1]\!]) \to [\![T_2]\!] = \\
\quad\quad \lambda(env : [\![f : T_1 \to T_2, \Gamma]\!], x : [\![T_1]\!]). \\
\quad\quad\quad \text{let } (f, x_1, \ldots, x_n) : [\![f : T_1 \to T_2, \Gamma]\!] = env \text{ in} \\
\quad\quad\quad [\![t]\!] \text{ in} \\
\quad \text{let rec } clo : [\![T_1 \to T_2]\!] = \\
\quad\quad \text{pack } [\![f : T_1 \to T_2, \Gamma]\!], (code, (clo, x_1, \ldots, x_n)) \\
\quad\quad \text{as} \\
\quad \text{in } clo
\end{array}
$$

Problem?

Environment-passing closure conversion

Here is how the "fix-pack" variant is type-checked. Assume $\Gamma \vdash \mu f.\lambda x.t : T_1 \to T_2$ and $dom(\Gamma) = \{x_1, \ldots, x_n\} = fv(\mu f.\lambda x.t)$.

$$
\begin{array}{l}
[\![\mu f : T_1 \to T_2.\lambda x.t]\!] = \\
\quad \text{let } code : ([\![f : T_1 \to T_2; \Gamma]\!] \times [\![T_1]\!]) \to [\![T_2]\!] = \\
\quad\quad \lambda(env : [\![f : T_1 \to T_2, \Gamma]\!], x : [\![T_1]\!]). \\
\quad\quad\quad \text{let } (f, x_1, \ldots, x_n) : [\![f : T_1 \to T_2, \Gamma]\!] = env \text{ in} \\
\quad\quad\quad [\![t]\!] \text{ in} \\
\quad \text{let rec } clo : [\![T_1 \to T_2]\!] = \\
\quad\quad \text{pack } [\![f : T_1 \to T_2, \Gamma]\!], (code, (clo, x_1, \ldots, x_n)) \\
\quad\quad \text{as } \exists X((X \times [\![T_1]\!]) \to [\![T_2]\!]) \times X) \\
\quad \text{in } clo
\end{array}
$$

Problem?

Environment-passing closure conversion

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

The problem is that polymorphic functions are indirectly compiled to polymorphic closures, by compiling the body to a monomorphic closure and generalizing afterward.

While this is fine without recursion or with monomorphic recursion, it does not work for polymorphic recursion.

Fortunately, the encoding can be easily adapted to directly compile a polymorphically recursive function into a polymorphic closure.

Environment-passing closure conversion

$$
\begin{array}{l}
[\![\mu f : \forall \vec{Y}.\, T_1 \to T_2.\, \lambda x.t]\!] = \\
\quad \text{let } code : \forall \vec{Y}.\, ([\![f : \forall \vec{Y}.\, T_1 \to T_2; \Gamma]\!] \times [\![T_1]\!]) \to [\![T_2]\!] = \\
\quad\quad \lambda(env : [\![f : \forall \vec{Y}.\, T_1 \to T_2, \Gamma]\!], x : [\![T_1]\!]). \\
\quad\quad\quad \text{let } (f, x_1, \ldots, x_n) : [\![f : \forall \vec{Y}.\, T_1 \to T_2, \Gamma]\!] = env \text{ in} \\
\quad\quad\quad [\![t]\!] \text{ in} \\
\quad \text{let rec } clo : [\![\forall \vec{Y}.\, T_1 \to T_2]\!] = \\
\quad\quad \Lambda \vec{Y}.\text{pack } [\![f : \forall \vec{Y}.\, T_1 \to T_2, \Gamma]\!], (code\ \vec{Y}, (clo, x_1, \ldots, x_n)) \\
\quad\quad\quad \text{as } \exists X((X \times [\![T_1]\!]) \to [\![T_2]\!]) \times X) \\
\quad \text{in } clo
\end{array}
$$

This is still simple.

However, introducing the construct "let rec x = v" requires altering the operational semantics and updating the type soundness proof.

Closure-passing closure conversion

Now, recall the *closure-passing* variant:

$$
\begin{aligned}
[\![\lambda x.\, t]\!] \quad = \quad & \text{let } code = \lambda(clo, x). \\
& \quad \text{let } (\_, x_1, \ldots, x_n) = clo \text{ in} \\
& \quad [\![t]\!] \\
& \text{in } (code, x_1, \ldots, x_n)
\end{aligned}
$$

$$
\begin{aligned}
[\![t_1\ t_2]\!] \quad = \quad & \text{let } clo = [\![t_1]\!] \text{ in} \\
& \text{let } code = \text{proj}_0\ clo \text{ in} \\
& code\ (clo, [\![t_2]\!])
\end{aligned}
$$

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

How could we typecheck this? What are the difficulties?

- 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

There are two difficulties:

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

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

Closure-passing closure conversion

There are two difficulties:

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

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

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

Tuples, rows, row variables

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

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

The notation $(T_1 \times \ldots \times T_n)$ was sugar for $\Pi\,(T_1; \ldots; T_n; \epsilon)$.

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

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

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:

$$\text{Tuple}\quad \frac{\forall i.\ \in [1,n] \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash (t_1,\ldots,t_n) : \Pi\,(T_1;\ldots;T_n;\epsilon)} \qquad \text{Proj}\quad \frac{\Gamma \vdash t : \Pi\,(T_1;\ldots;T_i;R)}{\Gamma \vdash \mathrm{proj}_i\ t : T_i}$$

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:*

$$\mathrm{proj}_i : \forall X._1 \ldots X_i \rho \Pi\,(X_1;\ldots;X_i;\rho) \to X_i$$

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:

$$
\begin{array}{lll}
 & [\![T_1 \to T_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
 & \quad \mu X. & X \text{ is the concrete type of the closure} \\
 & \quad\quad \Pi\,(& \text{a tuple...} \\
 & \quad\quad\quad (X \times [\![T_1]\!]) \to [\![T_2]\!]; & \text{...that begins with a code pointer...} \\
 & \quad\quad\quad \rho & \text{...and continues with the environment} \\
 & \quad\quad)
\end{array}
$$

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

Closure-passing closure conversion

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

$$
\begin{array}{lll}
 & [\![T_1 \to T_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
 & \quad \mu X. & X \text{ is the concrete type of the closure} \\
 & \quad\quad \Pi\,(& \text{a tuple...} \\
 & \quad\quad\quad (X \times [\![T_1]\!]) \to [\![T_2]\!]; & \text{...that begins with a code pointer...} \\
 & \quad\quad\quad \rho & \text{...and continues with the environment} \\
 & \quad\quad)
\end{array}
$$

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

Question: Why is it $\exists \rho.\, \mu X.\, T$ and not $\mu X.\, \exists \rho.\, T$

Closure-passing closure conversion

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

$$
\begin{array}{lll}
 & [\![T_1 \to T_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
 & \quad \mu X. & X \text{ is the concrete type of the closure} \\
 & \quad\quad \Pi\ (& \text{a tuple...} \\
 & \quad\quad\quad (X \times [\![T_1]\!]) \to [\![T_2]\!]; & \text{...that begins with a code pointer...} \\
 & \quad\quad\quad \rho & \text{...and continues with the environment} \\
 & \quad\quad)
\end{array}
$$

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

Question: Why is it $\exists \rho.\, \mu X.\, T$ and not $\mu X.\, \exists \rho.\, T$

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

Closure-passing closure conversion

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

$$
\begin{array}{lll}
 & [\![T_1 \to T_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
 & \quad \mu X. & X \text{ is the concrete type of the closure} \\
 & \quad\quad \Pi\,(& \text{a tuple...} \\
 & \quad\quad\quad (X \times [\![T_1]\!]) \to [\![T_2]\!]; & \text{...that begins with a code pointer...} \\
 & \quad\quad\quad \rho & \text{...and continues with the environment} \\
 & \quad\quad)
\end{array}
$$

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

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

Closure-passing closure conversion

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

$$
\begin{array}{lll}
 & [\![T_1 \to T_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
 & \quad \mu X. & X \text{ is the concrete type of the closure} \\
 & \quad\quad \Pi\,(& \text{a tuple...} \\
 & \quad\quad\quad (X \times [\![T_1]\!]) \to [\![T_2]\!]; & \text{...that begins with a code pointer...} \\
 & \quad\quad\quad \rho & \text{...and continues with the environment} \\
 & \quad\quad)
\end{array}
$$

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

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

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

Closure-passing closure conversion

Let $Clo(R)$ abbreviate $\mu X.\Pi\,((X \times [\![T_1]\!]) \to [\![T_2]\!]; R)$.

Let $UClo(R)$ abbreviate its unfolded version,
$\Pi\,((Clo(R) \times [\![T_1]\!]) \to [\![T_2]\!]; R)$.

We have $[\![T_1 \to T_2]\!] = \exists \rho.Clo(\rho)$.

$$
\begin{array}{rcl}
[\![\lambda x: \quad . t]\!] & = & \text{let } code : \qquad\qquad\qquad\qquad\qquad\qquad = \\
& & \quad \lambda(clo : \qquad\qquad , x : \qquad). \\
& & \quad\quad \text{let } (\_, x_1, \ldots, x_n) : \qquad\qquad = \text{unfold } clo \text{ in} \\
& & \quad\quad [\![t]\!] \text{ in} \\
& & \quad \text{pack} \qquad , (\text{fold } (code, x_1, \ldots, x_n)) \\
& & \quad \text{as} \\
\\
[\![t_1\ t_2]\!] & = & \text{let } \rho, clo = \text{unpack } [\![t_1]\!] \text{ in} \\
& & \quad \text{let } code : \qquad\qquad\qquad\qquad\qquad = \\
& & \quad\quad \text{proj}_0\ (\text{unfold } clo) \text{ in} \\
& & \quad code\ (clo, [\![t_2]\!])
\end{array}
$$

Closure-passing closure conversion

Let $Clo(R)$ abbreviate $\mu X.\Pi\,((X \times [\![T_1]\!]) \to [\![T_2]\!]; R)$.

Let $UClo(R)$ abbreviate its unfolded version,
$\Pi\,((Clo(R) \times [\![T_1]\!]) \to [\![T_2]\!]; R)$.

We have $[\![T_1 \to T_2]\!] = \exists \rho.Clo(\rho)$.

$$
\begin{array}{rcl}
[\![\lambda x{:}[\![T_1]\!].\, t]\!] & = & \mathsf{let}\ code : (Clo([\![\Gamma]\!]) \times [\![T_1]\!]) \to [\![T_2]\!] = \\
& & \quad \lambda(clo : Clo([\![\Gamma]\!], x : [\![T_1]\!]). \\
& & \quad\quad \mathsf{let}\ (\_, x_1, \ldots, x_n) : UClo[\![\Gamma]\!] = \mathsf{unfold}\ clo\ \mathsf{in} \\
& & \quad\quad [\![t]\!]\ \mathsf{in} \\
& & \mathsf{pack}\ [\![\Gamma]\!], (\mathsf{fold}\ (code, x_1, \ldots, x_n)) \\
& & \ \mathsf{as}\ \exists \rho.\, Clo(\rho)
\end{array}
$$

$$
\begin{array}{rcl}
[\![t_1\ t_2]\!] & = & \mathsf{let}\ \rho, clo = \mathsf{unpack}\ [\![t_1]\!]\ \mathsf{in} \\
& & \mathsf{let}\ code : (Clo(\rho) \times [\![T_1]\!]) \to [\![T_2]\!] = \\
& & \quad \mathsf{proj}_0\ (\mathsf{unfold}\ clo)\ \mathsf{in} \\
& & code\ (clo, [\![t_2]\!])
\end{array}
$$

Closure-passing closure conversion recursive functions

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

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

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

This untyped code can be easily typechecked—when recursion is monomorphic. — exercise!

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.

Closure-passing closure conversion

This encoding can not be adapted to to allow polymorphic recursion. The problem is that f is given the type of the closure when called, *i.e.* at some specialized type and not when defined at its polymorphic type.

There is not way this can work without changing the type of closures. $[\![T_1 \to T_2]\!]$.

Even changing $[\![T_1 \to T_2]\!]$, it does not seem possible to capture the typing constraints in System F.

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 T be $\forall \vec{X}.T_1 \to T_2$ and Γ_f be $f : T, \Gamma$ where $\vec{Y} \# \Gamma$

$$
\begin{array}{rcl}
[\![\mu f : T.\lambda x.t]\!] & = & \text{let } code = \\
& & \quad \Lambda\vec{Y}.\lambda(clo : Clo[\![\Gamma_f]\!], x : [\![T_1]\!]). \\
& & \quad\quad \text{let } (\_code, f, x_1, \ldots, x_n) : \forall\vec{Y}.UClo([\![\Gamma_f]\!]) = \\
& & \quad\quad\quad \text{unfold } clo \text{ in} \\
& & \quad\quad [\![t]\!] \text{ in} \\
& & \text{let rec } clo : \forall\vec{Y}.\exists\rho.Clo(\rho) = \Lambda\vec{Y}. \\
& & \quad \text{pack } [\![\Gamma]\!], (\text{fold } (code\ \vec{Y}, clo, x_1, \ldots, x_n)) \text{ as } \exists\rho.Clo(\rho) \\
& & \text{in } clo
\end{array}
$$

Remind that $Clo(R)$ abbreviates $\mu X.\Pi\,((X \times [\![T_1]\!]) \to [\![T_2]\!]; R)$. Hence, $\vec{Y}$ are free variables of $Clo(R)$.

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

Mutually recursive functions

Environment passing

Can we compile recursive functions?

$$t \triangleq \mu(f_1, f_2).(\lambda x_1.\, t_1, \lambda x_2.\, t_2)$$

Environment passing:

$$[\![t]\!] \quad =$$

Mutually recursive functions — Environment passing

Can we compile recursive functions?

$$t \triangleq \mu(f_1, f_2).(\lambda x_1.\, t_1, \lambda x_2.\, t_2)$$

Environment passing:

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

Mutually recursive functions — Environment passing

Can we compile recursive functions?

$$t \triangleq \mu(f_1, f_2).(\lambda x_1.\, t_1, \lambda x_2.\, t_2)$$

Environment passing:

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

Mutually recursive functions — Environment passing

Can we compile recursive functions?

$$t \triangleq \mu(f_1, f_2).(\lambda x_1.\, t_1, \lambda x_2.\, t_2)$$

Environment passing:

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

Comments?

Mutually recursive functions — Environment passing

Can we compile recursive functions?

$$t \triangleq \mu(f_1, f_2).(\lambda x_1.\, t_1, \lambda x_2.\, t_2)$$

Environment passing:

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

Mutually recursive functions

Closure passing

Encoding

$$
\begin{array}{l}
\text{let } code_i = \lambda(clo, x). \\
\quad \text{let } f_i = clo \text{ in} \\
\quad \text{let } (\_code, f_{3-i}, x_1, \ldots, x_n) = clo \text{ in } [\![t_i]\!] \\
\text{in} \\
\text{let rec } clo_1 = (code_1, clo_2, x_1, \ldots, x_n) \\
\quad \textit{and } clo_2 = (code_2, clo_1, x_1, \ldots, x_n) \\
\text{in } clo_1, clo_2
\end{array}
$$

Exercise: Is this still well-typed?

Question: Can we share the closures c_1 and c_2 in case n is large?

Can we merge the two encodings

Mutually recursive functions — Closure passing

$$
\begin{array}{l}
\text{let } code_1 = \lambda(clo, x). \\
\quad \text{let } (\_code_1, \_code_2, f_1, f_2, x_1, \ldots, x_n) = clo \text{ in } [\![t_1]\!] \text{ in} \\
\text{let } code_2 = \lambda(clo, x). \\
\quad \text{let } (\_code_2, f_1, f_2, x_1, \ldots, x_n) = clo \text{ in } [\![t_2]\!] \text{ in} \\
\text{let rec } clo_1 = (code_1, code_2, clo_1, clo_2, x_1, \ldots, x_n) \\
\quad \text{and } clo_2 = c_1.tail \\
\text{in } clo_1, clo_2
\end{array}
$$

Comments

- $clo_1.tail$ 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?

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:

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

$$
\begin{array}{rcl}
[\![t_1\ t_2]\!] & = & \mathsf{let}\ clo = [\![t_1]\!]\ \mathsf{in}\\
& & \mathsf{let}\ code = \mathsf{proj}_0\ clo\ \mathsf{in}\\
& & code\ (clo, [\![t_2]\!])
\end{array}
$$

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

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:

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

$$
\begin{array}{rcl}
[\![t_1\ t_2]\!] & = & \mathsf{let}\ clo = [\![t_1]\!]\ \mathsf{in}\\
& & \mathsf{let}\ code = \mathsf{proj}_0\ clo\ \mathsf{in}\\
& & code\ (clo, [\![t_2]\!])
\end{array}
$$

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

Encoding of objects

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

A class definition is an object generator:

$$
\begin{array}{l}
\mathit{class}\ c\ (x_1, \ldots x_q)\{ \\
\quad \mathrm{meth}\ m_1 = t_1 \\
\quad \ldots \\
\quad \mathrm{meth}\ m_p = t_p \\
\}
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{let}\ m = \\
\quad \mathsf{let}\ m_1 = \lambda(m, x_1, \ldots, x_q).\, t_1\ \mathsf{in} \\
\quad \ldots \\
\quad \mathsf{let}\ m_p = \lambda(m, x_1, \ldots, x_q).\, t_p\ \mathsf{in} \\
\quad \{m_1, \ldots, m_p\} \\
\lambda x_1 \ldots x_q.\, (m, x_1, \ldots x_q)
\end{array}
$$

Each m_i is bound to the code for corresponding method. All codes are combined into a record of codes.

Then, calling method m_i of an object p is

$$(\mathsf{proj}_0\ p).m_i\ p$$

How can we type the encoding?

Typed encoding of objects

Let T_i is the type of t_i, and row R describe the types of $(x_1, \ldots x_q)$.

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

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

$$
\begin{array}{l}
\mathit{let}\ m = \{ \\
\quad m_1 = \lambda(m, x_1, \ldots x_q : UClo(R)).\, t_1 \\
\quad \ldots \\
\quad m_p = \lambda(m, x_1, \ldots x_q : UClo(R)).\, t_p \\
\}\ \mathit{in} \\
\lambda x_1. \ldots \lambda x_q.\, \mathsf{pack}\ R, \mathsf{fold}\ (m, x_1, \ldots x_q)\ \mathsf{as}\ \exists \rho.\, (M, \rho)
\end{array}
$$

Calling a method of an object p of type M is

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

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].

Exercise: type-preserving CPS conversion

Here is an untyped version of call-by-value CPS conversion:

$$\begin{aligned}
[\![v]\!] &= \lambda k.\, k\ (\!|v|\!) \\
[\![t_1\ t_2]\!] &= \lambda k.\, [\![t_1]\!]\ (\lambda x_1.\, [\![t_2]\!]\ (\lambda x_2.\, x_1\ x_2\ k))
\end{aligned}$$

$$\begin{aligned}
(\!|x|\!) &= x \\
(\!|()|\!) &= () \\
(\!|(v_1, v_2)|\!) &= ((\!|v_1|\!), (\!|v_2|\!)) \\
(\!|\lambda x.\, t|\!) &= \lambda x.\, [\![t]\!]
\end{aligned}$$

Is this a type-preserving transformation?

The answer is in the 2007–2008 exam.

Another exercise

The 2006–2007 exam discusses a type-preserving translation of λ-calculus into bytecode.

Bibliography I

(Most titles have a clickable mark "▷" that links to online versions.)

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2): 78–102, March 1996.

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2–3): 81–116, December 1995.

▷ Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *ACM International Conference on Functional Programming (ICFP)*, pages 157–168, September 2008.

▷ Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

Bibliography II

- Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 38–49, January 2005.
- Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 54–65, June 2007.

Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–345. MIT Press, 2005.

- Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

Bibliography III

- John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.
- Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 63–74, January 2009.
- Greg Morrisett and Robert Harper. Typed closure conversion for recursively-defined functions (extended abstract). In *International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.
- Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

Bibliography IV

- François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.
- François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
- Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346. Springer, April 1994.
- Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.
- John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

Bibliography V

▷ Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1): 48–86, 1997.