2-4-2 / Type systems
Type-preserving closure conversion

François Pottier

October 27 and November 3, 2009

INRIA

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].

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.

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

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*.

Variants of closure conversion

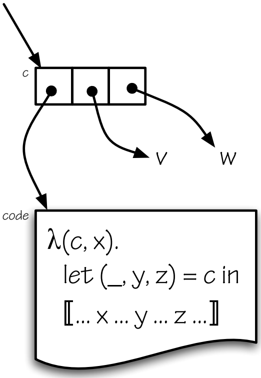
There are at least two variants of closure conversion:

- 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.

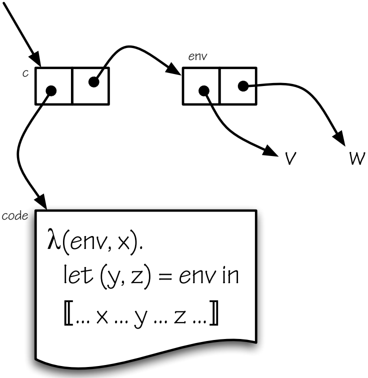
The impact of this choice on the term translations is minor.

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

Variants of closure conversion



closure-passing variant



environment-passing variant

let y = v and z = w in let c = λx.(... x ... y ... z ...) in ...

The closure-passing variant is as follows:

$$
\begin{array}{rcl}
[\![\lambda x.t]\!] & = & \text{let } code = \lambda(c, x). \\
& & \quad \text{let } (\_, x_1, \ldots, x_n) = c \text{ in} \\
& & \quad [\![t]\!] \\
& & \text{in } (code, x_1, \ldots, x_n) \\
\\
[\![t_1\ t_2]\!] & = & \text{let } c = [\![t_1]\!] \text{ in} \\
& & \text{let } code = \text{proj}_0\ c \text{ in} \\
& & code\ (c, [\![t_2]\!])
\end{array}
$$

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

Note that the layout of the environment must be known only at the closure allocation site, not at the call site.

(The variables *code* and *c* must be suitably fresh.)

The environment-passing variant is as follows:

$$
\begin{array}{rcl}
[\![\lambda x.t]\!] & = & \text{let } code = \lambda(env, x). \\
& & \quad \text{let } (x_1, \ldots, x_n) = env \text{ in} \\
& & \quad [\![t]\!] \\
& & \text{in } (code, (x_1, \ldots, x_n)) \\
[\![t_1\ t_2]\!] & = & \text{let } (code, env) = [\![t_1]\!] \text{ in} \\
& & code\ (env, [\![t_2]\!])
\end{array}
$$

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

Towards type-preserving closure conversion

Let us examine the closure allocation code again:

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

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

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 $([\![\Gamma]\!] \times [\![T_1]\!]) \rightarrow [\![T_2]\!]$, and the entire closure has type $(([\![\Gamma]\!] \times [\![T_1]\!]) \rightarrow [\![T_2]\!]) \times [\![\Gamma]\!]$.

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

(Parenthesis.)

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

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

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

Can we adopt this as a definition?

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

Can we adopt this as a definition?

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

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?

Yes, we do. *We need a uniform a 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 $[\![T_1 \rightarrow T_2]\!]$?*

The only sensible solution is:

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

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 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 *fully abstract:* it preserves (a typed version of) observational equivalence [Ahmed and Blume, 2008].

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.

Here is how the existential quantifier is introduced and eliminated:

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

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

$$\left(\text{TAbs}\quad \frac{\Gamma; X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T} \qquad \text{TApp}\quad \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t\ T' : [X \mapsto T']T} \right)$$

Note the (imperfect) *duality* between universals and existentials.

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

$$\frac{\Gamma \vdash t : \exists X.T \qquad X \# \Gamma}{\Gamma \vdash \mathsf{unpack}\; t : T}$$

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?

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

$$\frac{\Gamma \vdash t : \exists X.T \qquad X \# \Gamma}{\Gamma \vdash \text{unpack } t : T}$$

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 t has type $\forall X.T$. This is nonsense!

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.

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

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

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

Unpack

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

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

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

or:

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

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

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

Unpack

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

or:

Unpack

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

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

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

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 $X \# T_2$.

Pack

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

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

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

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

$$\begin{array}{rcl} \text{pack}_{\exists X.T} & : & \forall X.(T \to \exists X.T) \\ \text{unpack}_{\exists X.T} & : & \exists X.T \to \forall Y.((\forall X.(T \to Y)) \to Y) \end{array}$$

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? What is it?

The type translation is *double negation:*

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

The term translation is:

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

The type translation is *double negation:*

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

The term translation is:

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

The type translation is *double negation:*

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

The term translation is:

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

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

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

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.

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:

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

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

I cut a few corners on the previous slide. The "type scheme:"

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

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

I cut a few corners 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:

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

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 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:

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

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*.

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:

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

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

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

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

(Recall that $Y \;\#\; T_2$.)

Define

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

A client that regards Y as abstract succeeds:

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

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

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

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.

Everything is now set up to prove that

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

Environment-passing closure conversion

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

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

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

Assume $\Gamma \vdash t_1 : T_1 \to T_2$ and $\Gamma \vdash t_2 : T_1$.

$$
\begin{aligned}
[\![t_1\ t_2]\!] \quad = \quad & \text{let } X, (code, env) = \text{unpack } [\![t_1]\!] \text{ in} && code : (X \times [\![T_1]\!]) \to [\![T_2]\!] \\
& code\ (env, [\![t_2]\!]) && env : X \\
& && (X \ \#\ [\![T_2]\!])
\end{aligned}
$$

We find $[\![\Gamma]\!] \vdash [\![t_1\ t_2]\!] : [\![T_2]\!]$, as desired.

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

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

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?

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

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

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

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.

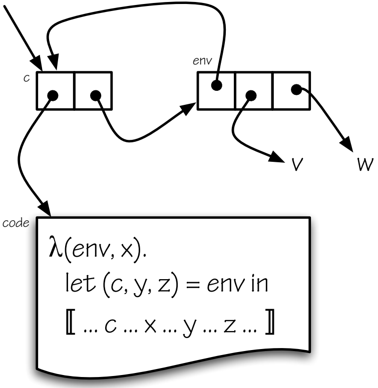
Instead, the "fix-pack" variant [Morrisett and Harper, 1998] uses an extra field in the environment to store a back pointer to the closure:

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

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

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

Environment-passing closure conversion



environment-passing variant, fix-pack

let y = v and z = w in let rec c = λx.(... c ... x ... y ... z ...) in ...

Here is how the "fix-pack" variant is type-checked.

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

let *code* = $\lambda(\mathit{env}, x)$.	$\mathit{env} : [\![f : T_1 \rightarrow T_2; \Gamma]\!]; x : [\![T_1]\!]$
let $(f, x_1, \ldots, x_n) = \mathit{env}$ in	this installs $[\![f : T_1 \rightarrow T_2; \Gamma]\!]$
$[\![t]\!]$	$[\![t]\!] : [\![T_2]\!]$
in	$\mathit{code} : ([\![f : T_1 \rightarrow T_2; \Gamma]\!] \times [\![T_1]\!]) \rightarrow [\![T_2]\!]$
let rec *c* =	now, assume $c : [\![T_1 \rightarrow T_2]\!]$
pack $(\mathit{code}, (c, x_1, \ldots, x_n))$	this has type $[\![T_1 \rightarrow T_2]\!]$ too...
in *c*	so all is well

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

Now, recall the *closure-passing* variant:

$$
\begin{array}{rcl}
[\![\lambda x.t]\!] & = & \text{let } code = \lambda(c, x). \\
& & \quad \text{let } (\_, x_1, \ldots, x_n) = c \text{ in} \\
& & \quad [\![t]\!] \\
& & \text{in } (code, x_1, \ldots, x_n) \\
[\![t_1\ t_2]\!] & = & \text{let } c = [\![t_1]\!] \text{ in} \\
& & \text{let } code = \text{proj}_0\ c \text{ in} \\
& & code\ (c, [\![t_2]\!])
\end{array}
$$

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

How could we typecheck this? What are the difficulties?

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?

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.*

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.

The typing rules for tuple construction and deconstruction are:

Tuple
$$\frac{\forall i \in [1, n] \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash (t_1, \ldots, t_n) : \Pi\ (T_1; \ldots; T_n; \epsilon)}$$

Proj
$$\frac{\Gamma \vdash t : \Pi\ (T_1; \ldots; T_i; R)}{\Gamma \vdash \mathrm{proj}_i\ t : T_i}$$

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:*

$$\mathrm{proj}_i : \forall X_1 \ldots X_i \rho. \Pi\ (X_1; \ldots; X_i; \rho) \longrightarrow X_i$$

Rows were invented independently by Wand and 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].

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

$[\![T_1 \rightarrow T_2]\!]$	
$= \exists\rho.$	ρ describes the environment
$\quad \mu X.$	X is the concrete type of the closure
$\quad\quad \Pi\ ($	a tuple...
$\quad\quad\quad (X \times [\![T_1]\!]) \rightarrow [\![T_2]\!];$	...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].

Closure-passing closure conversion

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

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

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

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

- $c : Clo([\![\Gamma]\!]); x : [\![T_1]\!]$
- this installs $[\![\Gamma]\!]$
- $code : (Clo([\![\Gamma]\!]) \times [\![T_1]\!]) \rightarrow [\![T_2]\!]$
- the tuple has type $UClo([\![\Gamma]\!])$
- after folding, $Clo([\![\Gamma]\!])$
- after packing, $[\![T_1 \rightarrow T_2]\!]$

$$
\begin{aligned}
[\![t_1\; t_2]\!] = \;& \text{let } c, \rho = \text{unpack } [\![t_1]\!] \text{ in} \\
& \text{let } code = \text{proj}_0 \text{ (unfold } c) \text{ in} \\
& code\; (c, [\![t_2]\!])
\end{aligned}
$$

- $c : Clo(\rho)$
- $code : (Clo(\rho) \times [\![T_1]\!]) \rightarrow [\![T_2]\!]$
- this has type $[\![T_2]\!]$

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

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

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

Again, this untyped code can be typechecked. — *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.

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.

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].

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 is a type-preserving transformation?

The answer is in the 2007–2008 exam.

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

(Most titles are clickable links to online versions.)

Ahmed, A. and Blume, M. 2008.
Typed closure conversion preserves observational equivalence.
In *ACM International Conference on Functional Programming (ICFP)*. 157–168.

Chen, J. and Tarditi, D. 2005.
A simple typed intermediate language for object-oriented languages.
In *ACM Symposium on Principles of Programming Languages (POPL)*. 38–49.

Bibliography]Bibliography

Chlipala, A. 2007.
A certified type-preserving compiler from lambda calculus to assembly language.
In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 54–65.

Harper, R. and Pierce, B. C. 2005.
Design considerations for ML-style module systems.
In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, Chapter 8, 293–345.

[][

Läufer, K. and Odersky, M. 1994.
Polymorphic type inference and abstract data types.
ACM Transactions on Programming Languages and Systems 16, 5 (Sept.), 1411–1430.

Mitchell, J. C. and Plotkin, G. D. 1988.
Abstract types have existential type.
ACM Transactions on Programming Languages and Systems 10, 3, 470–502.

Montagu, B. and Rémy, D. 2009.
Modeling abstract types in modules with open existential types.
In *ACM Symposium on Principles of Programming Languages (POPL)*. 63–74.

[][

Morrisett, G. and Harper, R. 1998.
Typed closure conversion for recursively-defined functions (extended abstract).
In *International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*. Electronic Notes in Theoretical Computer Science, vol. 10. Elsevier Science.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999.
From system F to typed assembly language.
ACM Transactions on Programming Languages and Systems 21, 3 (May), 528–569.

[][

Pottier, F. and Gauthier, N. 2006.
Polymorphic typed defunctionalization and concretization.
Higher-Order and Symbolic Computation 19, 125–162.

Pottier, F. and Rémy, D. 2005.
The essence of ML type inference.
In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, Chapter 10, 389–489.

Rémy, D. and Vouillon, J. 1998.
Objective ML: An effective object-oriented extension to ML.
Theory and Practice of Object Systems 4, 1, 27–50.

[][

Reynolds, J. C. 1983.
Types, abstraction and parametric polymorphism.
In *Information Processing 83*. Elsevier Science, 513–523.

Steckler, P. A. and Wand, M. 1997.
Lightweight closure conversion.
ACM Transactions on Programming Languages and Systems 19, 1, 48–86.