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

François Pottier

January 22, 2008

R INRIA

Compilation is type-preserving compilation 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

There are at least two variants of closure conversion:

- in the *closure-passing variant,* presented earlier in this course by Xavier Leroy, 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.

The closure-passing variant is as follows:

$$
\begin{aligned}
[\![\lambda x.t]\!] \;=\; & \text{let } code = \lambda(c, x). \\
& \qquad \text{let } (\_, x_1, \ldots, x_n) = c \text{ in} \\
& \qquad [\![t]\!] \\
& \text{in } (code, x_1, \ldots, x_n) \\[1em]
[\![t_1 \; t_2]\!] \;=\; & \text{let } c = [\![t_1]\!] \text{ in} \\
& \text{let } code = \text{proj}_0 \, c \text{ in} \\
& code \, (c, [\![t_2]\!])
\end{aligned}
$$

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:

$$
\begin{aligned}
[\![\lambda x.t]\!] \;\; = \;\; & \text{let } code = \lambda(env, x). \\
& \qquad \text{let } (x_1, \ldots, x_n) = env \text{ in} \\
& \qquad [\![t]\!] \\
& \text{in } (code, (x_1, \ldots, x_n)) \\[1em]
[\![t_1 \; t_2]\!] \;\; = \;\; & \text{let } (code, env) = [\![t_1]\!] \text{ in} \\
& code \, (env, [\![t_2]\!])
\end{aligned}
$$

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

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

Let us examine the closure allocation code again:

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

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 $\llbracket \Gamma \rrbracket$ for the tuple type $T_1 \times \ldots \times T_n$.

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

Now, *what should be the definition of $\llbracket T_1 \to T_2 \rrbracket$?*

(Parenthesis.)

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

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

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?

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

Can we adopt this as a definition?

$$\llbracket T_1 \to T_2 \rrbracket \quad = \quad ((\llbracket \Gamma \rrbracket \times \llbracket T_1 \rrbracket) \to \llbracket T_2 \rrbracket) \times \llbracket \Gamma \rrbracket$$

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:

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

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:

$$\llbracket \mathcal{T}_1 \to \mathcal{T}_2 \rrbracket \quad = \quad \exists X.((X \times \llbracket \mathcal{T}_1 \rrbracket) \to \llbracket \mathcal{T}_2 \rrbracket) \times X$$

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 remain 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.

Contents

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 explicit and implicit variants of the terms and typing rules... Let's just look at the explicit variant.

In the explicit variant, there are term-level constructs for introducing and eliminating the existential quantifier:

Pack

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

Unpack

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

$$\left(\begin{array}{cc} \text{TAbs} & \text{TApp} \\ \dfrac{\Gamma; X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T} & \dfrac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t \ T' : [X \mapsto T']T} \end{array} \right)$$

Note the *duality* between universals and existentials. A somewhat imperfect duality, since existentials have a rather complex elimination form...

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?

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!

Removing the premise $X \# \Gamma$ would make the rule even more permissive, so it wouldn't help.

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 on the *user:* in the rule, the term t_2.

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

The restriction $X \# T_2$ prevents writing "let $X, x = $ 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 *bla-bla-bla-bla* in X...

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

Unpack

$$\frac{\Gamma \vdash t_1 : \exists X.T_1 \qquad X \# T_2}{\Gamma \vdash \Lambda X.\lambda x.t_2 : \forall X.T_1 \rightarrow T_2}$$

$\Gamma \vdash \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2$

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

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:

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

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

or:

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

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 $X \# T_2$.

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:

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

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 extract a Y out of it, then you must present a function which, for *any* X, extracts 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:*

$$\llbracket \exists X.T \rrbracket \quad = \quad \forall Y.((\forall X.(\llbracket T \rrbracket \to Y)) \to Y) \qquad \text{if } Y \# T$$

The term translation is:

$$
\begin{aligned}
\llbracket \text{pack}_{\exists X.T} \rrbracket \quad &: \quad \forall X.(\llbracket T \rrbracket \to \llbracket \exists X.T \rrbracket) \\
&= \quad ? \\
\llbracket \text{unpack}_{\exists X.T} \rrbracket \quad &: \quad \llbracket \exists X.T \rrbracket \to \forall Y.((\forall X.(\llbracket T \rrbracket \to Y)) \to Y) \\
&= \quad ?
\end{aligned}
$$

The type translation is *double negation:*

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

The term translation is:

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

The type translation is *double negation:*

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

The term translation is:

$$
\begin{aligned}
[\![pack_{\exists X.T}]\!] \quad &: \quad \forall X.([\![T]\!] \rightarrow [\![\exists X.T]\!]) \\
&= \quad \wedge X.\lambda x : [\![T]\!].\wedge Y.\lambda f : \forall X.([\![T]\!] \rightarrow Y).f\ X\ x \\
[\![unpack_{\exists X.T}]\!] \quad &: \quad [\![\exists X.T]\!] \rightarrow \forall Y.((\forall X.([\![T]\!] \rightarrow Y)) \rightarrow Y) \\
&= \quad \lambda x : [\![\exists X.T]\!].x
\end{aligned}
$$

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

What is the computational content of this encoding?

The type translation is *double negation:*

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

The term translation is:

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

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

What is the computational content of this encoding?

A *continuation-passing transform.*

This encoding is due to Reynolds [1983].

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*:

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

This introduces two constants, with the following type schemes:

$$\text{pack}_D \ : \ \forall \bar{X}\bar{Y}.T \rightarrow D \ \vec{X}$$
$$\text{unpack}_D \ : \ \forall \bar{X}Z.D \ \vec{X} \rightarrow (\forall \bar{Y}.(T \rightarrow Z)) \rightarrow Z$$

(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:"

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

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

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

Unpack$_D$

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

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:

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

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:

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

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

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

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

(Recall that $Y \# T_2$.)

Define
$$D \ X_1 \ X_2 \approx \exists Y.((Y \times X_1) \to X_2) \times Y$$

A client that regards Y as abstract succeeds:

$$
\begin{aligned}
& \llbracket \text{unpack}_D \ t_1 \ (\lambda(f, y).f \ (y, x)) : T_2 \rrbracket \\
=\ & \exists X_1 X_2.(\llbracket t_1 : D \ X_1 \ X_2 \rrbracket \wedge \forall Y. \llbracket \lambda(f, y).f \ (y, x) : (((Y \times X_1) \to X_2) \times Y) \to T_2 \rrbracket) \\
\equiv\ & \exists X_1 X_2.(\llbracket t_1 : D \ X_1 \ X_2 \rrbracket \wedge \forall Y.\text{def} \ f : (Y \times X_1) \to X_2; y : Y \text{ in } \llbracket f \ (y, x) : T_2 \rrbracket) \\
\equiv\ & \exists X_1 X_2.(\llbracket t_1 : D \ X_1 \ X_2 \rrbracket \wedge \forall Y.(x \preceq X_1 \wedge T_2 = X_2)) \\
\equiv\ & \exists X_1 X_2.(\llbracket t_1 : D \ X_1 \ X_2 \rrbracket \wedge x \preceq X_1 \wedge T_2 = X_2) \\
\equiv\ & \exists X_1.(\llbracket t_1 : D \ X_1 \ T_2 \rrbracket \wedge x \preceq X_1)
\end{aligned}
$$

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

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

specifies an abstract implementation of integer stacks.

Unfortunately, it was soon noticed that the elimination rule is *too awkward*, and that existential types alone do not allow designing *module systems* (ATTAPL, Chapter 8).

Contents

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

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

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

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

$$[\![t_1\ t_2]\!] \quad = \quad \text{let } X, (code, env) = \text{unpack } [\![t_1]\!] \text{ in} \qquad code : (X \times [\![T_1]\!]) \rightarrow [\![T_2]\!]$$
$$code\ (env, [\![t_2]\!]) \qquad\qquad\qquad\qquad env : X$$

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

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

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

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

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

Another approach, known as the "fix-pack"
variant [Morrisett and Harper, 1998], allocates an extra field in the
environment to store a back pointer to the closure:

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

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

Recall that the closure-passing variant is as follows:

$$\llbracket \lambda x.t \rrbracket \quad = \quad \text{let } code = \lambda(c, x).$$
$$\text{let } (\_, x_1, \ldots, x_n) = c \text{ in}$$
$$\llbracket t \rrbracket$$
$$\text{in } (code, x_1, \ldots, x_n)$$

$$\llbracket t_1 \ t_2 \rrbracket \quad = \quad \text{let } c = \llbracket t_1 \rrbracket \text{ in}$$
$$\text{let } code = \text{proj}_O \ c \text{ in}$$
$$code \ (c, \llbracket t_2 \rrbracket)$$

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 syntax of types is extended:

$$
\begin{aligned}
T &::= \quad \dots \mid \Pi \, R \mid \exists \rho.T \mid \mu X.T \\
R &::= \quad \dots \mid \rho \mid \epsilon \mid (T; R)
\end{aligned}
$$

The corresponding typing rules are omitted.

The type translation is now somewhat more involved:

$$\llbracket T_1 \to T_2 \rrbracket$$
$$= \exists \rho. \qquad \qquad \qquad \rho \text{ describes the environment}$$
$$\mu X. \qquad \qquad X \text{ is the concrete type of the closure}$$
$$\Pi \left((X \times \llbracket T_1 \rrbracket) \to \llbracket T_2 \rrbracket ; \rho \right) \quad \text{a tuple that begins with the code pointer}$$
$$\text{and continues with the environment}$$

We can now typecheck this code: — not written yet! (exercise)

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

$$\llbracket t_1 \; t_2 \rrbracket \;=\; \begin{aligned}[t] &\text{let } c = \llbracket t_1 \rrbracket \text{ in} \\ &\text{let } code = \text{proj}_0 \; c \text{ in} \\ &code \; (c, \llbracket t_2 \rrbracket) \end{aligned}$$

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

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

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

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

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.

Xavier's presentation uses a *pun* on the names f and c, which amounts to statically reducing the highlighted let-redex above.

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.

Contents

(Most titles are clickable links to online versions.)

📄 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.

📄 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.

Bibliography]Bibliography

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.

[ ][

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