Type systems for programming languages

Didier Rémy

Academic year 2014-2015
Version of December 18, 2014
CONTENTS

5.3.3 Constraint solving by example ................................................. 103
5.3.4 Type reconstruction .............................................................. 106
5.4 Type annotations ................................................................. 109
  5.4.1 Explicit binding of type variables ........................................ 110
  5.4.2 Polymorphic recursion ....................................................... 113
  5.4.3 mixed-prefix .................................................................. 114
5.5 Equi- and iso-recursive types .................................................... 115
  5.5.1 Equi-recursive types .......................................................... 115
  5.5.2 Iso-recursive types ............................................................ 117
  5.5.3 Algebraic data types .......................................................... 118
5.6 \textit{HM(X)} ........................................................................ 119
5.7 Type reconstruction in System \textit{F} ................................................. 121
  5.7.1 Type inference based on Second-order unification .................. 121
  5.7.2 Bidirectional type inference ................................................. 122
  5.7.3 Partial type inference in \textit{MLF} ........................................... 124
5.8 Proofs and Solution to Exercises ............................................... 124

6 Existential types ................................................................. 127
  6.1 Towards typed closure conversion ........................................... 128
  6.2 Existential types ................................................................. 130
    6.2.1 Existential types in Church style (explicitly typed) .............. 130
    6.2.2 Implicitly-type existential types ....................................... 133
    6.2.3 Existential types in \textit{ML} .............................................. 135
    6.2.4 Existential types in \textit{OCaml} ......................................... 136
  6.3 Typed closure conversion ...................................................... 137
    6.3.1 Environment-passing closure conversion ............................ 137
    6.3.2 Closure-passing closure conversion .................................. 139
    6.3.3 Mutually recursive functions ......................................... 141

7 Logical Relations ................................................................. 145
  7.1 Introduction ................................................................. 145
  7.2 Normalization of simply-typed \(\lambda\)-calculus ......................... 145
  7.3 Proofs and Solution to Exercises ........................................... 147

8 Overloading ................................................................. 149
  8.1 An overview ................................................................. 149
    8.1.1 Why use overloading? .................................................... 149
    8.1.2 Different forms of overloading ........................................ 150
    8.1.3 Static overloading ......................................................... 151
    8.1.4 Dynamic resolution with a type passing semantics ............. 151
    8.1.5 Dynamic overloading with a type erasing semantics .......... 152
Chapter 6

Existential types

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.

Type preserving compilation is interesting for several reasons: it can help debug the compiler; types can be used to drive optimizations; types can also be used to produce *proof-carrying code*; proving that types are preserved during compilation can be the first step towards proving that the *semantics* is preserved [Chlipala (2007)].

Besides, type-preserving compilation is quite challenging as it exhibits an encoding of programming constructs into programming language that usually requires richer type systems. Sometimes, an encoding later becomes a programming idiom that is used directly in the source language. There are several examples: closure conversion requires an extension of the language with existential types, which happens to very useful on their own. Closures are themselves a simple form of objects. Defunctionalization may be done manually on some particular programs, *e.g.* in web applications to monitor the computation.

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

1. *CPS conversion* fixes the order of evaluation, names intermediate computations, and makes all function calls tail calls;

2. *closure conversion* makes environments and closures explicit, and produces a program where all functions are closed;

3. allocation and initialization of tuples is made explicit;

4. 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 $M$ to $\llbracket M \rrbracket$, but also a translation of types, mapping $\tau$ to $\llbracket \tau \rrbracket$, with the property:

$$ \Gamma \vdash M : \tau \implies [\Gamma] \vdash [M] : [\tau] $$

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

### 6.1 Towards typed closure conversion

First-class functions may appear in the body of other functions, hence, their own body may contain free variables that will be bound to values during the evaluation in the execution environment. Because they can be returned as values, and thus used outside of their definition environment, they must store their execution environment in their value. A closure is the packaging of the code of a first-class function with its runtime environment, so that it becomes closed, i.e. independent of the runtime environment and can be passed to another function 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.

In the following, the source calculus has unary $\lambda$-abstractions, which can have free variables, while the target calculus has binary $\lambda$-abstractions, which must be closed. In the target language, we also use pattern matching over tuples. The translation will be naive, insofar as it will not handle functions of multiple arguments in a special way. One could argue that this is a feature, not a limitation, and that “uncurrying” (if desired) should be a separate type-preserving pass anyway. But closure conversion can also be easily extended to n-ary functions.

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. Closure-passing better supports simple recursive functions; but this is less obvious with mutually recursive ones. Closure-passing optimizes the case of closed functions: they is no need to create a closure—the code pointer can be passed directly Steckler and Wand [1997]. However, its impact on the type translations is more important: the closure-passing variant requires more type-theoretic machinery (recursive types and rows).

The closure-passing variant is as follows:

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

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

$$ [M_1 M_2] = \text{let } clo = [M_1] \text{ in } $$

$$ \text{let } code = \text{proj}_0 clo \text{ in } $$

$$ \text{code (clo, [M_2])} $$
where \( \{ x_1, \ldots, x_n \} \) is \( \text{fv}(\lambda x. M) \) (the variables \textit{code} and \textit{clo} must be suitably fresh). Note that the layout of the environment must be known only at the closure allocation site, not at the call site. In particular, \( \text{proj}_0 \text{clo} \) need not know the size of \textit{clo}.

The environment-passing variant is as follows:

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

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

To understand type-preserving closure conversion, let us first focus on the environment-passing variant. How can closure conversion be made \textit{type-preserving}? The key issue is to find a sensible definition of the type translation. In particular, what is the translation of a function type, \( \llbracket \tau_1 \rightarrow \tau_2 \rrbracket \)? Let us examine the closure allocation code again. Suppose \( \Gamma \vdash \lambda x. M : \tau_1 \rightarrow \tau_2 \). Suppose, without loss of generality (see Remark 3), that \( \text{dom} (\Gamma) \) is exactly \( \text{fv}(\lambda x. M) \), i.e. \( \{ x_1, \ldots, x_n \} \). Overloading the notation, if \( \Gamma \) is \( x_1 : \tau_1; \ldots; x_n : \tau_n \), we also write \( [\Gamma] \) for the tuple type \( [\tau_1] \times \ldots \times [\tau_n] \). By hypothesis, we have \( [\Gamma], x : [\tau_1] \vdash [M] : [\tau_2] \), so \text{env} has type \( [\Gamma] \), \text{code} has type \( ([\Gamma] \times [\tau_1]) \rightarrow [\tau_2] \), and the entire closure has type \( (([\Gamma] \times [\tau_1]) \rightarrow [\tau_2]) \times [\Gamma] \).

So, can we adopt \( (([\Gamma] \times [\tau_1]) \rightarrow [\tau_2]) \times [\Gamma] \) as a definition of \( \llbracket \tau_1 \rightarrow \tau_2 \rrbracket \)?

Naturally not. This definition is mathematically ill-formed, as we cannot use \( \Gamma \) out of the blue! That is, we cannot have a translation of \( \llbracket \tau_1 \rightarrow \tau_2 \rrbracket \) that depends on the type of free variables of \( M \)! Indeed, we need a \textit{uniform translation of types}, not just because it is nice to have one, but because it describes a \textit{uniform calling convention}. If closures with distinct environment sizes or layouts receive distinct types, then we will be unable to translate well-typed code: 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, the only sensible solution is: \( \exists \alpha. (\alpha \times [\tau_1]) \rightarrow [\tau_2]) \times \alpha \). An \textit{existential quantification} over the type of the environment abstracts away the differences in size and layout. Enough information is retained to ensure that the application of the code to the environment is valid: this is expressed by letting the variable \( \alpha \) occur twice on the right-hand side.

The existential quantification also provides a form of \textit{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 \textit{well-typed} client is unable to do so. This encoding is not just type-preserving, but also \textit{fully abstract}: it preserves (a typed
version of) observational equivalence (Ahmed and Blume, 2008).

**Remark 5** In order to support the hypothesis \( \text{dom}(\Gamma) = \text{fv}(\lambda x. M) \) at every \( \lambda \)-abstraction, it is possible to introduce an (admissible) *weakening* rule:

\[
\frac{\text{WEAKENING}}{
\Gamma_1; \Gamma_2 \vdash M : \tau \quad x \not\in M}
\quad \Gamma_1; x : \tau'; \Gamma_2 \vdash M : \tau
\]

If the weakening rule is applied eagerly at every \( \lambda \)-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.)

### 6.2 Existential types

One can extend System \( \text{F} \) with *existential types*, in addition to universals:

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

As in the case of universals, there are *type-passing* and *type-erasing* interpretations of the terms and typing rules and, in the latter interpretation, there are *explicit* and *implicit* versions. Let us first look at the type-erasing interpretation with an explicit notation for introducing and eliminating existential types.

#### 6.2.1 Existential types in Church style (explicitly typed)

The existential quantifier are introduced and eliminated as follows:

\[
\frac{\text{PACK}}{
\Gamma \vdash M : [\alpha \mapsto \tau']\tau}
\quad \frac{\text{UNPACK}}{
\Gamma \vdash M_1 : \exists \alpha. \tau_1 \quad \Gamma, \alpha, x : \tau_1 \vdash M_2 : \tau_2}
\quad \alpha \not\in \tau_2
\]

\[
\Gamma \vdash \text{pack} \tau', M \text{ as } \exists \alpha. \tau : \exists \alpha. \tau
\quad \Gamma \vdash \text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2 : \tau_2
\]

The side condition \( \alpha \not\in \tau_2 \) is *mandatory* here to ensure well-formedness of the conclusion. If well-formedness conditions were explicit in judgments, this could be equivalently defined as \( \Gamma \vdash \tau_2 \), as it would imply \( \alpha \not\in \Gamma \) since the last premise implies \( \alpha \not\in \Gamma \).

Notice the *imperfect* duality between existential and universals, reminded below:

\[
\frac{\text{TAbs}}{
\Gamma, \alpha \vdash M : \tau}
\quad \frac{\text{TApp}}{
\Gamma \vdash M : \forall \alpha. \tau}
\]

\[
\Gamma \vdash \Lambda \alpha. M : \forall \alpha. \tau
\quad \Gamma \vdash M \tau' : [\alpha \mapsto \tau']\tau
\]

This suggests a simpler elimination form, perhaps like this:

\[
\Gamma \vdash M : \exists \alpha. \tau
\]

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

*Broken!*
Informally, this could mean that, if $M$ has type $\tau$ for some unknown $\alpha$, then it has type $\tau$, where $\alpha$ is “fresh”. Unfortunately, this is a broken rule, as we could immediately universally quantify over $\alpha$ and conclude that $\Gamma \vdash M : \forall \alpha. \tau$. This is nonsense! Replacing the premise $\Gamma, \alpha \vdash M : \exists \alpha. \tau$ by the conjunction $\Gamma \vdash M : \exists \alpha. \tau$ and $\alpha \in \text{dom}(\Gamma)$ would make the rule even more permissive, so it wouldn’t help.

A correct elimination rule must force the existential package to be used in a way that does not rely on the value of $\alpha$. Hence, the elimination rule must have control over the user or continuation of the package—that is, over the term $M_2$. The restriction $\alpha \not\# \tau_2$ prevents writing “let $\alpha, x = \text{unpack} M_1$ in $x$”, which would be equivalent to the unsound “unpack $M$” discussed above. The fact that $\alpha$ is bound within $M_2$ forces it to be treated abstractly. In fact, $M_2$ must be polymorphic in $\alpha$. The rule could be written:

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

Or, more economically:

$$\Gamma \vdash M_1 : \exists \alpha. \tau_1 \quad \Gamma \vdash M_0 : \forall \alpha. \tau_1 \to \tau_2 \quad \alpha \not\# \tau_2$$

where $M_0$ would evaluate to a value of the form $\Lambda \alpha. \lambda x. M_2$.

One could even view “unpack$_{\exists \alpha. \tau}$” as a constant, of type $\exists \alpha. \tau \to \forall \beta. ((\forall \alpha. (\tau \to \beta)) \to \beta)$. The variable $\beta$, which stands for $\tau_2$, is bound prior to $\alpha$, so it naturally cannot be instantiated to a type that refers to $\alpha$. This reflects the side condition $\alpha \not\# \tau_2$. If desired, “pack$_{\exists \alpha. \tau}$” could also be viewed as a constant of type $\forall \alpha. (\tau \to \exists \alpha. \tau)$.

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

$$\text{pack}_{\exists \alpha. \tau} : \forall \alpha. (\tau \to \exists \alpha. \tau) \quad \text{unpack}_{\exists \alpha. \tau} : \exists \alpha. \tau \to \forall \beta. ((\forall \alpha. (\tau \to \beta)) \to \beta) \quad (\Delta_\exists)$$

These can be read as follows: for any $\alpha$, if you have a $\tau$, then, for some $\alpha$, you have a $\tau$; conversely, if, for some $\alpha$, you have a $\tau$, then, (for any $\beta$) if you wish to obtain a $\beta$ out of $\exists \alpha. \tau$, you must present a function which, for any $\alpha$, obtains a $\beta$ out of a $\tau$. This is somewhat reminiscent of ordinary first-order logic: $\exists x. F$ is equivalent to, and can be defined as, $\neg(\forall x. \neg F)$.

One can go one step further and entirely encode existential types into universal types. This encoding is actually a small example of type-preserving translation! The type translation is double negation:

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

There is actually little choice for the term translation, if the translation is to be type-
preserving:

\[
[\text{pack}_{\exists \alpha. \tau}] : \forall \alpha. ([\tau] \to [\exists \alpha. \tau])
= \Lambda \alpha. \lambda x : [\tau]. \Lambda \beta. \lambda k : \forall \alpha. ([\tau] \to \beta). k \alpha x
\]

\[
[\text{unpack}_{\exists \alpha. \tau}] : [[\exists \alpha. \tau] \to \forall \alpha. ([\tau] \to \beta) \to \beta)
= \lambda x : [[\exists \alpha. \tau]]. x
\]

This encoding is a *continuation-passing transform*. This encoding is due to Reynolds [1983], although it has more ancient roots in logic.

When existential are presented as constants, their semantics is defined by seeing \(\text{pack}_{\exists \alpha. \tau}\) as a unary constructor and \(\text{unpack}_{\exists \alpha. \tau}\) as a unary destructor with the following reduction rule:

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

(\(\delta_3\))

**Exercise 42** Show that this \(\delta\)-rule satisfies the progress and subject reduction assumptions for constants with the types in \(\Delta_3\). (You may assume that the standard lemmas still hold.)

(Solution p. 173)

**Exercise 43** The \(\delta_3\) reduction for existential is permissive it allows reducing of ill-typed terms. Give a more restrictive version of the rule. What will need to be changed in the proof of subject reduction and process for the \(\delta\)-rule (Exercise 42)?

(Solution p. 173)

Notice that our \(\delta_3\)-reduction reduces an “unpack of a pack” to a polymorphic function that applies its argument to the packed value. This is still a form of continuation-passing-style encoding. It seems more natural to treat \(\text{unpack}_{\exists \alpha. \tau}\) as a binary destructor to avoid this intermediate step and have the more intuitive reduction rule:

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

(\(\delta_3\))

However, this does not fit in our framework and notion of arity for constants where all type arguments must be passed first and not interleaved with value arguments. Our framework could be extended to the above \(\delta\)-rules for existentials, but the presentation would become cumbersome.

Alternatively, if existential are primitive, their semantics is defined by extending values and evaluation contexts as follows:

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

and by adding the following reduction rule:

\[
\text{let } \alpha, x = \text{unpack} (\text{pack} \tau', V \text{ as } \tau) \text{ in } M \rightarrow [\alpha \mapsto \tau'] [x \mapsto V] M
\]
Exercise 44  Check that the proofs of subject reduction and progress for System F extend to existential types. (Just check the new cases, assuming that the standard lemmas still hold.) □

The reduction rule for existential destructs its arguments. Hence, let α, x = unpack M₁ in M₂ cannot be reduced unless M₁ is itself a packed expression, which is indeed the case when M₁ is a value (or in head normal form). This contrasts with let x : τ = M₁ in M₂ where M₁ need not be evaluated and may be an application (e.g. in call-by-name or with strong reduction).

Exercise 45  The reduction of let α, x = unpack M₁ in M₂ could be problematic when M₁ is not a value. Illustrate this on an example (You may use the following hint if needed: lanoitidnocesu.) (Solution p. 174)

One may wonder whether the pack construct is not too verbose: isn’t the type witness type annotation τ’ in rule PACK superfluous? The type τ₀ of M is fully determined by M and the given type ∃α. τ of the packed value. Checking that τ₀ is of the form [α ↦ τ’]τ is the matching problem for second-order types, which is simple. However, the reduction rule need the witness type τ’. 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 by second-order matching.

6.2.2 Implicitly-type existential types

Intuitively, pack and unpack are just type information that can be dropped by type erasure. More precisely, the erasure of pack τ’, M as ∃α. τ ∃α. τ is M and the erasure of let α, x = unpack M₁ in M₂ is a let-binding let x = M₁ in M₂. After type-erasure, the following typing rules for existential types in implicit-typed System F:

\[
\begin{align*}
\text{if-Unpack} & : \quad \Gamma \vdash a₁ : ∃α. τ₁ \quad \Gamma, α, x : τ₁ \vdash a₂ : τ₂ \quad α \# τ₂ \\
\Gamma & \vdash \text{let } x = a₁ \text{ in } a₂ : τ₂
\end{align*}
\]

Notice, that the let-binding is not typechecked as syntactic sugar for an immediate application. Its semantics remains the same.

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

Is the semantics still type-erasing? Yes, it is, but there is a subtlety! This is only true in call-by-value. In a call-by-name semantics, a let-bound expression is not reduced prior to substitution of the argument, that is, the rule would be:

\[\text{let } x = a₁ \text{ in } a₂ \rightarrow [x \mapsto a₁]a₂\]

With existential types, this breaks subject reduction!
To see this, let $\tau_0$ be $\exists \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha)$ and let $v_0$ be a value of type $\text{bool}$. Then, let $v_1$ and $v_2$ two values of type $\tau_0$ with incompatible witness types, taking for instance, $\lambda f. \lambda x. 1+ (f (1 + x))$ and $\lambda f. \lambda x. \text{not} (f (\text{not} x))$. Let $v$ be the function $\lambda b. \text{if}\ b\ \text{then}\ v_1\ \text{else}\ v_2$ of type $\text{bool} \to \tau_0$, which returns either one of $V_1$ or $V_2$ depending on its argument $b$. We then have the reduction

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

The typing judgment $\emptyset \vdash a_1 : \exists \alpha. \alpha \to \alpha$ holds, while $\emptyset \vdash a_2 : \tau$ does not hold for any $\tau$. Indeed, the term $a_1$ is well-typed since $v \ v_0$ has type $\tau_0$, hence $x$ can be assumed of type $(\beta \to \beta) \to (\beta \to \beta)$ for some unknown type $\beta$ and $\lambda y. y$ is of type $\beta \to \beta$. However, without the outer existential type $v \ v_0$ can only be typed with $(\forall \alpha. \alpha \to \alpha) \to \exists \alpha. (\alpha \to \alpha)$, because the value returned by the function need different witnesses for $\alpha$. This is demanding too much on its argument and the outer application is ill-typed.

One may wonder whether the syntax should not allow the implicit introduction of unpacking instead. For instance, one could argue that if some expression is the expansion of a well-typed let-binding, then it should also be well-typed:

$$
\Gamma \vdash a_1 : \exists \alpha. \tau_1 \quad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \quad \alpha \neq \tau_2

\frac{}{\Gamma \vdash [x \mapsto a_1]a_2 : \tau_2}
$$

However, this rule is not quite satisfactory as it does not have a logical flavor. Moreover, it fixes the previous example, but does not help with 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 $[x \mapsto a_1]a_2$.

In summary, existential types are tricky: The subject reduction property breaks if reduction is not restricted to expressions in head-normal forms. Unrestricted reduction is still safe because well-typedness may eventually be recovered by further reduction steps—so that progress will never break.

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

$$\llbracket \text{unpack} \ a_1 \ (\lambda x. a_2) \rrbracket = \llbracket a_1 \rrbracket (\lambda x. \llbracket a_2 \rrbracket) \quad (1)
\quad \to \quad (\lambda k. \llbracket a \rrbracket \ k) (\lambda x. \llbracket a_2 \rrbracket) \quad (2)
\quad \to \quad (\lambda x. \llbracket a_2 \rrbracket) \llbracket a \rrbracket \quad (3)
\quad \to \quad [x \mapsto \llbracket a \rrbracket] \llbracket a \rrbracket \quad (4)$$

In the call-by-value setting, $\lambda k. \llbracket a \rrbracket \ k$ would come from the reduction of $\llbracket \text{pack} \ a \rrbracket$, i.e. is $(\lambda k. \lambda x. k \ x) \llbracket a \rrbracket$, so that $a$ is always a value $v$. However, $a$ need not be a value. What is essential is again that $a_1$ be reduced to some head normal form $\lambda k. \llbracket a \rrbracket \ k$. 
6.2. EXISTENTIAL TYPES

6.2.3 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.

This iso-existential approach was suggested by Läüfer and Odersky (1994). Iso-existential types are explicitly declared, much as datatypes:

\[ D \vec{\alpha} \approx \exists \vec{\beta}. \tau \quad \text{if ftv}(\tau) \subseteq \vec{\alpha} \cup \vec{\beta} \quad \text{and} \quad \vec{\alpha} \# \vec{\beta} \]

This introduces two constants, with the following type schemes:

\[ \text{pack}_D : \forall \vec{\alpha}\vec{\beta}. \tau \to D \vec{\alpha} \quad \text{unpack}_D : \forall \vec{\alpha}\gamma. D \vec{\alpha} \to (\forall \vec{\beta}. (\tau \to \gamma)) \to \gamma \]

(Compare with basic iso-recursive types, where \( \vec{\beta} = \varnothing \).)

Unfortunately, the "type scheme" of \( \text{unpack}_D \) is not an ML type scheme. A solution is to make \( \text{unpack}_D \) a binary primitive construct, rather than a constant, with an ad hoc typing rule:

\[
\begin{align*}
\text{UNPACK}_D \\
\Gamma \vdash M_1 : D \vec{\tau} &\quad \Gamma \vdash M_2 : \forall \vec{\beta}. ([\vec{\alpha} \mapsto \vec{\tau}] \tau \to \tau_2) \quad \vec{\beta} \# \vec{\tau}, \tau_2 \\
\Gamma \vdash \text{unpack}_D M_1 M_2 : \tau_2 &\quad \text{where } D \vec{\alpha} \approx \exists \vec{\beta}. \tau
\end{align*}
\]

We have seen a version of this rule in System F earlier; this in an ML version. The term \( M_2 \) must be polymorphic, which \text{GEN} can prove.

Iso-existential types are perfectly compatible with ML type inference. The constant \( \text{pack}_D \) admits an ML type scheme, so it is not problematic. The construct \( \text{unpack}_D \) leads to this constraint generation rule (cf. §5):

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

where \( D \vec{\alpha} \approx \exists \vec{\beta}. \tau \) and, w.l.o.g., \( \vec{\alpha} \vec{\beta} \# M_1, M_2, \tau_2 \). Note that a universally quantified constraint appears where polymorphism is required.

In practice, Läüfer and Odersky suggest fusing iso-existential types with algebraic data types. The somewhat bizarre Haskell syntax for this is:

\[
data D \vec{\alpha} = \forall \vec{\beta}. \ell \tau
\]

where \( \ell \) is a data constructor. The elimination construct \( \langle \text{case } M_1 \text{ of } \ell \ x \to M_2 : \tau_2 \rangle \) and is typed as follows:

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

where, w.l.o.g., \( \vec{\alpha} \vec{\beta} \# M_1, M_2, \tau_2 \).
Examples Define \( \text{Any} \approx \exists \beta. \beta \). The following code that attempts to extract the raw content of a package fails:

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

Now, define \( D \alpha \approx \exists \beta. (\beta \to \alpha) \times \beta \). A client that regards \( \beta \) as abstract succeeds:

\[
\langle \text{unpack} \ D \ M_1 \ (\lambda (f, y). \ f \ y) : \tau \rangle = \exists \alpha. (\langle M_1 : D \alpha \rangle \land \forall \beta. \langle \lambda (f, y). \ f \ y : ((\beta \to \alpha) \times \beta) \to \tau \rangle)
\]

Remark 6 We reuse the type \( D \alpha \approx \exists \beta. (\beta \to \alpha) \times \beta \) of frozen computations, defined above. Assume given a list \( l \) of elements of type \( D \tau_1 \). Assume given a function \( g \) of type \( \tau_1 \to \tau_2 \).

We may transform the list into a new list \( l' \) of frozen computations of type \( D \tau_2 \) (without actually running any computation).

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

We may generalize the code into a functional that receives \( g \) and \( l \) as arguments and returns \( l' \). Unfortunately, the following code does not typecheck:

\[
\text{let} \ \text{lif}\text{t} \ g \ l = \text{List.map} \ (\lambda (z) \ \text{let} \ D (f, y) = z \ \text{in} \ D ((\lambda (z) \ g (f \ z)), y))
\]

The problem is that, in expression \( \text{let} \ \alpha, x = \text{unpack} \ M_1 \ \text{in} \ M_2 \), occurrences of \( x \) can only be passed to polymorphic functions so that the type \( \alpha \) of \( x \) does not escape from its scope. That is first-class existential types calls for first-class universal types as well!

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

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

specifies an abstract implementation of integer stacks.

Unfortunately, it was soon noticed that the elimination rule is too awkward, and that existential types alone do not allow designing module systems Harper and Pierce (2005). Montagu and Rémy (2009) make existential types more flexible in several important ways, and argue that they might explain modules after all.

6.2.4 Existential types in OCaml

Amusingly, existential types were first available in OCaml via abstract types and first-class modules. There are now also available as a degenerate case of Generalized Algebraic DataTypes (GADT) which coincides with the approach described above.

For example, one may define the previous datatype of frozen computations:
Here is the equivalent, more verbose code with modules:

```ocaml
module type D = sig type a val f : b → a val x : b end
let freeze (type u) (type v) f x =
  (module struct type b = u type a = v let f = f let x = x end : D);;
let unfreeze (type u) (module M : D with type a = u) = M.f M.x
```

### 6.3 Typed closure conversion

Equipped with existential types, we may now revisit type closure conversion.

#### 6.3.1 Environment-passing closure conversion

Remember that we came to the conclusion that the translation of arrow types $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket$ must be $\exists \alpha.((\alpha \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket) \times \alpha$. Let us show that we may translate expressions so as to preserve well-typedness, i.e. so that $\Gamma \vdash M : \tau$ implies $\llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket \tau \rrbracket$. Assume $\Gamma \vdash \lambda x. M : \tau_1 \rightarrow \tau_2$ and $\text{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \text{fv}(\lambda x. M)$. We may now hide the dependence on $\Gamma$ using an existential type:

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

In the case of application, assume $\Gamma \vdash M : \tau_1 \rightarrow \tau_2$ and $\Gamma \vdash M_1 : \tau_1$ and take:

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

For recursive functions we may use the “fix-code” variant (Morrisett and Harper, 1998):

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

where $\{x_1, \ldots, x_n\} = \text{fv}(\mu f . \lambda x. M)$. The translation of applications is unchanged as recursive and non-recursive functions have an identical calling convention. This translation builds recursive code, avoiding a recursive closure, hence the code is easy to type. Unfortunately, as a counterpart, a new closure is allocated at every call, which is the weak point of this variant.
Instead, the “fix-pack” variant (Morrisett and Harper, 1998) uses an extra field in the environment to store a back pointer to the closure:

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

where \( \{x_1, \ldots, x_n\} = \text{fv}(\mu f.\lambda x.M) \). Hence, we avoid rebuilding the closure at every call by creating a recursive closure. However, this requires, in general, recursively-defined \emph{values} and closures are now \emph{cyclic} data structures.

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

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

This implements monomorphic recursion, as by default in ML. To allow the recursive function to be polymorphic, we can generalize the encoding afterwards:

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

whenever the right-hand side is well-defined. This allows the \emph{indirect} compilation of polymorphic recursive functions as long as the recursion is monomorphic.

Fortunately, the encoding can be straightforwardly adapted to \emph{directly} compile polymorphically recursive functions into polymorphic closure.

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

In summary, the environment-passing closure conversion is simple, but it requires the introduction of recursive non-functional values \( \text{let rec } x = V \text{ in } M \). While this is a useful construct, it really alters the operational semantics and requires updating the type soundness proof (as recursive non-functional values were not permitted so far).
6.3. Closure-passing closure conversion

Recall the closure-passing variant:

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

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

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

There are two difficulties to typecheck this: first, a closure is a tuple, whose first field—the code pointer—should be exposed, while the number and types of the remaining fields—the environment—should be abstract; second, the first field of the closure contains a function that expects the closure itself as its first argument.

To describe this, we use two type-theoretic mechanisms; first existential quantification over the tail of a tuple (a.k.a. a row) to allow the environment to remain abstract; and recursive types to allow the closure to points to itself.

Tuples, rows, row variables Let us first introduce extensible tuples. The standard tuple types that we have used so far are:

\[
\tau ::= \ldots | \Pi R \quad \text{– types}
\]

\[
R ::= \epsilon | (\tau; R) \quad \text{– rows}
\]

The notation \((\tau_1 \times \ldots \times \tau_n)\) was sugar for \(\Pi (\tau_1; \ldots; \tau_n; \epsilon)\). Let us introduce row variables and allow quantification over them:

\[
\tau ::= \ldots | \Pi R | \forall \rho. \tau | \exists \rho. \tau \quad \text{– types}
\]

\[
R ::= \rho | \epsilon | (\tau; R) \quad \text{– rows}
\]

This allows reasoning about the first few fields of a tuple whose length is not known. The typing rules for tuple construction and deconstruction are:

\[
\text{TUPLE}
\]

\[
\Gamma \vdash M_i : \tau_i \\
\begin{array}{c}
\forall i. \epsilon [1, n] \\
\end{array}
\]

\[
\Gamma \vdash (M_1, \ldots, M_n) : \Pi (\tau_1; \ldots; \tau_n; \epsilon)
\]

\[
\text{PROJ}
\]

\[
\Gamma \vdash M : \Pi (\tau_1; \ldots; \tau_i; R) \\
\begin{array}{c}
\text{proj}_i
\end{array}
\]

\[
\Gamma \vdash \text{proj}_i M : \tau_i
\]

These rules make sense with or without row variables. Projection does not care about the fields beyond \(i\). Thanks to row variables, this can be expressed in terms of parametric polymorphism: \(\text{proj}_i : \forall \alpha_1 \ldots \alpha_i \rho. \Pi (\alpha_1; \ldots; \alpha_i; \rho) \rightarrow \alpha_i\).

Remark 7 Rows were invented by Wand (1988) and improved by Rémy (1994b) 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 OCaml (Rémy and Vouillon, 1998). Rows are explained in depth by Pottier and Rémy (2005).
Back to closure-passing closure conversion

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

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

\( \rho \) describes the environment represented as a row of fields, which is abstract; \( \alpha \) is the concrete type of the closure that is to refer to recursively; \( \Pi \left( ((\alpha \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket) ; \rho \right) \) is a tuple that begins with a code pointer of type \( (\alpha \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket \) and continues with the environment \( \rho \). See the “fix-type” encoding proposed by Morrisett and Harper (1998).

Notice that the type is \( \exists \rho. \mu \alpha. \tau \) and not \( \mu \alpha. \exists \rho. \tau \): The type of the environment is fixed once for all and does not change at each recursive call. Notice that \( \rho \) appears only once, which may seem surprising. Usually, an existential type variable appears both at positive and negative occurrences. Here, the variable \( \alpha \) appear only at a negative occurrence, but in a recursive part of the type that can be unfolded.

To help checking well-typedness of the encoding, let \( \text{Clo}(R) \) abbreviate the concrete type of a closure of row \( R \) and \( \text{UClo}(R) \) its unfolded version:

\[
\text{Clo}(R) \triangleq \mu \alpha. \Pi \left( (\alpha \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket ; R \right) \\
\text{UClo}(R) \triangleq \Pi \left( (\text{Clo}(R) \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket ; R \right)
\]

The encoding of arrow types \( \llbracket \tau_1 \to \tau_2 \rrbracket \) is \( \exists \rho. \text{Clo}(\rho) \). The encoding of abstractions and applications is:

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

\[
\llbracket M_1 M_2 \rrbracket = \text{let } \rho, clo = \text{unpack } \llbracket M_1 \rrbracket \text{ in} \\
\text{let code : (\text{Clo}(\rho) \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket = proj_0 (\text{unfold clo) in}} \\
\text{code (clo, } \llbracket M_2 \rrbracket)\]

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

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

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

where \( \{ x_1 , \ldots , x_n \} = \text{fv}(\mu f . \lambda x . M) \). No extra field or extra work is required to store or construct a representation of the free variable \( f \): the closure itself plays this role. However, this untyped code can only be typechecked when recursion is monomorphic.

Exercise 46 Carefully check well-typedness of the above translation with monomorphic recursion.

\[ \square \]
6.3. TYPED CLOSURE CONVERSION

To adapt this encoding to polymorphic recursion, the problem is that recursive occurrences of $f$ are rebuilt from the current invocation of the closure, this with the same type since the closure is invoked after type specialization.

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

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

**Remark 8** One could think of changing the encoding of closure types $[\tau_1 \to \tau_2]$ to make the encoding work. However, although this should be possible in some more expressive type systems, there seems to be no easy way to do so and certainly not within System $\mathcal{F}$.

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

$$[\mu f : \tau. \lambda x. M] = \text{let code } = \Lambda \beta. \lambda (clo : \text{Clo}[\Gamma_f], x : [\tau_1]).$$

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

$$\text{let rec } clo : \forall \beta. \exists \rho. \text{Clo}(\rho) = \Lambda \beta. \text{pack } [\Gamma], (\text{fold } (\text{code } \beta, clo, x_1, \ldots, x_n)) \text{ as } \exists \rho. \text{Clo}(\rho) \text{ in } clo$$

Remind that $\text{Clo}(R)$ abbreviates $\mu \alpha. \Pi ((\alpha \times [\tau_1]) \to [\tau_2]; R)$. Hence, $\beta$ are free variables of $\text{Clo}(R)$. Here, a polymorphic recursive function is directly compiled into a polymorphic recursive closure. Notice that the type of closures is unchanged, so the encoding of applications is also unchanged.

**Optimizing representations** Closure-passing and environment-passing closure conversions cannot be mixed because the calling-convention (i.e., the encoding of application) must be uniform. However, there is some flexibility in the representation of the closure. For instance, the following change is completely local:

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

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

This allows for sharing the closure (or part of it) may be shared when many definitions share the same closure,

6.3.3 Mutually recursive functions

Can we compile mutually recursive functions $\mu(f_1, f_2), (\lambda x_1. M_1, \lambda x_2. M_2)$, say $M$?
The environment passing encoding is as follows:

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

Notice that we can share the environment inside the two closures. The closure passing encoding is:

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

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

Here the environment cannot be shared between the two closures, since they belong to tuples of different size. Unless the runtime, in particular the garbage collector, supports such an operation as returning the tail of a tuple without allocating a new tuple. Then we could write:

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

Here \( \text{clo}_1.\text{tail} \) returns a pointer to the tail \((\text{code}_2, \text{clo}_1, \text{clo}_2, x_1, \ldots, x_n)\) of \( \text{clo}_1 \) without allocating a new tuple.

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

A class definition is an object generator:

\[
\text{class } c \ (x_1, \ldots x_q) \ \{ \text{meth } m_1 = M_1; \ldots \text{meth } m_q = M_q \}\]

Given arguments for parameter \( x_1, \ldots x_n \), it builds recursive methods \( m_1, \ldots m_n \). A class can be compiled into an object closure:

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

Each \( m_i \) is bound to the code for the corresponding method. All codes are combined into a
record of codes. Then, calling method $m_i$ of an object $p$ is $(\text{proj}_0 p).m_i p$.

Let us write the typed version of this encoding. Let $\tau_i$ be the type of $M_i$ and row $R$ describe the types of $(x_1, \ldots, x_q)$. Let $\text{Clo}(R)$ be $\mu \alpha. \Pi((m_i : \alpha \to \tau_i)^{i=1..n}; R)$ and $\text{UClo}(R)$ its unfolding.

Fields $R$ are hidden in an existential type $\mu \alpha. \Pi((m_i : \alpha \to \tau_i)^{i=1..n}; \rho)$:

\[
\begin{align*}
\text{let } m &= \\
&\{ m_1 = \lambda(m, x_1, \ldots, x_q : \text{UClo}(R)).[M_1]; \\
&\vdots \\
&m_p = \lambda(m, x_1, \ldots, x_q : \text{UClo}(R)).[M_p] \} \text{ in}\\
&\lambda x_1. \ldots \lambda x_q. \text{pack } R, \text{fold } (m, x_1, \ldots, x_q) \text{ as } \exists \rho. (M, \rho)
\end{align*}
\]

Calling a method of an object $p$ of type $M$ is

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

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

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 (1994a) 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.

Summary

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 naive 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.

Defunctionalization, an alternative to closure conversion, offers an interesting challenge, with a simple solution. See, for instance 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).

One may think that references in System $F$ could be translated away by making the store explicit. In fact, this can be done, but not in System $F$, nor even in System $F^\omega$: the translation is quite tricky and in order for the translation to be well-typed the type system must be reach enough to express monotonicity of the store in a context where the store is itself recursively defined. See Pottier (2011) for details.
Exercise 47 (CPS conversion) Here is an untyped version of call-by-value CPS conversion:

\[
[V] = \lambda k. k \langle V \rangle
\]

\[
[M_1 M_2] = \lambda k. [M_1] (\lambda x_1. [M_2] (\lambda x_2. x_1 x_2 k))
\]

\[
\langle x \rangle = x
\]

\[
\langle () \rangle = ()
\]

\[
\langle (V_1, V_2) \rangle = (\langle V_1 \rangle, \langle V_2 \rangle)
\]

\[
\langle \lambda x. M \rangle = \lambda x. [M]
\]

Is this a type-preserving transformation? (Solution p. 174)
Appendix A

Proofs and Answers to exercises

Solution of Exercise 42

We first need to show that the $\delta_3$ preserves typings. Assume that

$$\Gamma \vdash \text{unpack}_{3a,\tau} (\text{pack}_{3a,\tau} \tau' V) : \tau_0$$

By inversion of typing, $\tau_1$ and $\tau_0$ must be equal to $\tau$ and $\forall \beta. (\forall \alpha. \tau \rightarrow \beta) \rightarrow \beta$, respectively, and the judgment $\Gamma \vdash V : [\alpha \mapsto \tau']_\tau$ must hold. Let $\Gamma'$ be $\Gamma, \beta, y : \forall \alpha. \tau \rightarrow \beta$. By weakening, we have $\Gamma' \vdash V : [\alpha \mapsto \tau']_\tau$. We then have $\Gamma' \vdash y \tau' V : \beta$ and finally, we have

$$\Gamma \vdash \Lambda \beta. \lambda y : \forall \alpha. \tau \rightarrow \beta. y \tau' V : \tau_0$$

as expected.

We then need to show that $\delta_3$ satisfies progress, i.e., a full well typed application of $\text{unpack}_{3a,\tau}$ can always be reduced. Assume that $\Gamma \vdash \text{unpack}_{3a,\tau} V : \tau_0$. By inversion of typing, we must have $\Gamma \vdash V : 3a. \tau$. By the classification lemma (to be extended and rechecked), $V$ must be an existential value, i.e., of the form $\text{pack}_{3a,\tau} \tau_0 V_0$. Hence, $\text{unpack}_{3a,\tau} V$ reduces by $\delta_3$.

Solution of Exercise 43

We just force $\tau_1$ to coincide with $\tau$:

$$\text{unpack}_{3a,\tau} (\text{pack}_{3a,\tau} \tau' V) \rightarrow \Lambda \beta. \lambda y : \forall \alpha. \tau \rightarrow \beta. y \tau' V$$

($\delta_3$)

The proof of subject reduction will know by construction that $\tau_0$ is $\tau$ instead of learning it by inversion of typing. Conversely for progress, we will have to show that $\tau_1$ and $\tau$ are equal by inversion so that $\delta_3$ can be applied.
Solution of Exercise 45

Let $M_1$ be if $M$ then $V_1$ else $V_2$ where $V_i$ is of the form pack $\tau_i$, $V_i$ as $\exists \alpha \tau$ and the two witnesses $\tau_1$ and $\tau_2$ differ. There is no common type for the unpacking of the two possible results $V_1$ and $V_2$. The choice between those two possible results must be made, by evaluating $M_1$, before unpacking.

Solution of Exercise 47

The answer is in the 2007–2008 exam.
Bibliography

▷ A tour of scala: Implicit parameters. Part of scala documentation.


