Type systems for programming languages

Didier Rémy

Academic year 2014-2015
Version of January 2, 2017

Contents

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 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 $[\![M]\!]$, but also a translation of *types,* mapping τ to $[\![\tau]\!]$, with the property:

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

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

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* λ-abstractions, which can have free variables, while the *target* calculus has *binary* λ-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:

$$\begin{array}{rcl}
[\![\lambda x.\, a]\!] & = & \mathsf{let}\ code = \lambda(clo,\, x).\, \mathsf{let}\ (_, x_1, \ldots, x_n) = clo\ \mathsf{in}\ [\![a]\!]\ \mathsf{in} \\
 & & (code, x_1, \ldots, x_n) \\[1ex]
[\![a_1\ a_2]\!] & = & \mathsf{let}\ clo = [\![a_1]\!]\ \mathsf{in} \\
 & & \mathsf{let}\ code = \mathsf{proj}_0\ clo\ \mathsf{in} \\
 & & code\ (clo,\, [\![a_2]\!])
\end{array}$$

where $\{x_1, \ldots, x_n\}$ is $\mathsf{fv}(\lambda x.\, a)$ (the variables *code* and *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, $\mathsf{proj}_0\ clo$ need not know the size of *clo*.

The environment-passing variant is as follows:

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

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

To understand 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, $[\![\tau_1 \to \tau_2]\!]$? Let us examine the closure allocation code again. Suppose $\Gamma \vdash \lambda x.\, a : \tau_1 \to \tau_2$. Suppose, without loss of generality (see Remark 5), that $\mathsf{dom}(\Gamma)$ is exactly $\mathsf{fv}(\lambda x.\, a)$, *i.e.* $\{x_1, \ldots, x_n\}$. If Γ is $x_1 : \tau'_1; \ldots; x_n : \tau'_n$, we write $[\![\Gamma]\!]$ for $x_1 : [\![\tau'_1]\!]; \ldots; x_n : [\![\tau'_n]\!]$. By abuse of notation, we also use $[\![\Gamma]\!]$ in a type position to mean the tuple type $[\![\tau'_1]\!] \times \ldots \times [\![\tau'_n]\!]$.

By hypothesis, we have $[\![\Gamma]\!], x : [\![\tau_1]\!] \vdash [\![a]\!] : [\![\tau_2]\!]$, so *env* has type $[\![\Gamma]\!]$, *code* has type $([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!]$, and the entire closure has type $(([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times [\![\Gamma]\!]$. So, can we adopt $(([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times [\![\Gamma]\!]$ as a definition of $[\![\tau_1 \to \tau_2]\!]$?

Naturally not. This definition is mathematically ill-formed, as we cannot use Γ out of the blue! That is, we cannot have a translation of $[\![\tau_1 \to \tau_2]\!]$ that depends on the type of free variables of a! Indeed. *we need a uniform translation of types,* not just because it is nice to have one, but because it describes a *uniform calling convention.* If closures with distinct environment sizes or layouts receive distinct types, then we will be unable to translate well-typed code: $\mathsf{if}\ \ldots\ \mathsf{then}\ \lambda x.\, x + y\ \mathsf{else}\ \lambda x.\, x$. Furthermore, we want function invocations to be translated uniformly, without knowledge of the size and layout of the closure's environment.

The only sensible solution is: $\exists\alpha.((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha$. An *existential quantification* over the type of the environment abstracts away the differences in size and layout. Enough information is retained to ensure that the application of the code to the environment is valid: this is expressed by letting the variable α occur twice on the right-hand side.

The 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:

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

After closure conversion, the reference x is reachable via the closure of f. A malicious, untyped client could write an odd value to x. However, a *well-typed* client is unable to do so. This encoding is not just type-preserving, but also *fully abstract:* it preserves (a typed version of) observational equivalence (Ahmed and Blume, 2008).

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

$$\frac{\Gamma_1; \Gamma_2 \vdash a : \tau \qquad x \# a}{\Gamma_1; x : \tau'; \Gamma_2 \vdash a : \tau}\ \textsc{Weakening}$$

If the weakening rule is applied eagerly at every λ-abstraction, then the hypothesis is met, and closures have *minimal environments.* (In some cases, one may not use minimal environments, *e.g.* to allow sharing of environments between several closures.)

6.2 Existential types

One can extend System 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:

$$\frac{\Gamma \vdash M : [\alpha \mapsto \tau']\tau}{\Gamma \vdash \mathsf{pack}\ \tau', M\ \mathsf{as}\ \exists\alpha.\,\tau : \exists\alpha.\,\tau}\ \textsc{Pack} \qquad \frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma, \alpha, x : \tau_1 \vdash M_2 : \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2 : \tau_2}\ \textsc{Unpack}$$

The side condition $\alpha \# \tau_2$ 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 $\alpha \# \tau_2$ since the last premise implies $\alpha \# \Gamma$.

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

$$\frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\,\tau}\ \textsc{TAbs} \qquad \frac{\Gamma \vdash M : \forall\alpha.\,\tau}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}\ \textsc{TApp}$$

This suggests a simpler elimination form, perhaps like this:

$$\frac{\Gamma \vdash M : \exists\alpha.\tau}{\Gamma, \alpha \vdash \mathsf{unpack}\ M : \tau}$$

Broken!

Informally, this could mean that, if M has type τ for some *unknown* α, then it has type τ, where α is "fresh". Unfortunately, this is a broken rule, as we could immediately *universally* quantify over α 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 $\alpha \in \mathsf{dom}(\Gamma)$ 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 α. Hence, the elimination rule must have control over the *user* or *continuation* of the package—that is, over the term M_2. The restriction $\alpha \# \tau_2$ prevents writing "$\mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ x$", which would be equivalent to the unsound "$\mathsf{unpack}\ M$" discussed above. The fact that α is bound within M_2 forces it to be treated abstractly. In fact, M_2 must be *polymorphic* in α. The rule could be written:

$$\frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma \vdash \Lambda\alpha.\lambda x.\, M_2 : \forall\alpha.\, \tau_1 \to \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2 : \tau_2}$$

Or, more economically:

$$\frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma \vdash M_0 : \forall\alpha.\, \tau_1 \to \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \mathsf{unpack}\ M_1\ M_0 : \tau_2}$$

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

One could even view "unpack" as a *constant* with all the types $(\exists\alpha.\tau_1) \to (\forall\alpha.\,(\tau_1 \to \tau_2)) \to \tau_2$. or, letting β range over τ_2, all types $\forall\beta.\,(\exists\alpha.\tau) \to (\forall\alpha.\,(\tau \to \beta)) \to \beta$ or even better, $\exists\alpha.\tau \to \forall\beta.\,((\forall\alpha.\,(\tau \to \beta)) \to \beta)$, since β should not occur free in τ. We thus introduce a *family* of constants "$\mathsf{unpack}_{\exists\alpha.\tau}$" with type $\exists\alpha.\tau \to \forall\beta.\,((\forall\alpha.\,(\tau \to \beta)) \to \beta)$. Notice that the variable β, which stands for τ_2, is bound prior to α, so it naturally cannot be instantiated to a type that refers to α. This reflects the side condition $\alpha \# \tau_2$. If desired, "$\mathsf{pack}_{\exists\alpha.\tau}$" could also be viewed as a constant of type $\forall\alpha.\,(\tau \to \exists\alpha.\tau)$. Similarly, we may introduce a constant pack with all the types $[\alpha \mapsto \tau']\tau \to \exists\alpha.\tau$, which we may factor as the following types $\forall\alpha.\,(\tau \to \exists\alpha.\tau)$.

In summary, System F with existential types can also be presented by introducing two families of constants of constants with the following types:

$$\mathsf{pack}_{\exists\alpha.\tau} :\ \forall\alpha.\,(\tau \to \exists\alpha.\tau) \qquad \mathsf{unpack}_{\exists\alpha.\tau} :\ \exists\alpha.\,\tau \to \forall\beta.\,((\forall\alpha.\,(\tau \to \beta)) \to \beta) \qquad (\Delta_\exists)$$

These can be read as follows: for *any* α, if you have a τ, then, for *some* α, you have a τ; conversely, if, for *some* α, you have a τ, then, (for any β,) if you wish to obtain a β out of $\exists\alpha.\,\tau$, you must present a function which, for *any* α, obtains a β out of a τ. 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:*

$$[\![\exists\alpha.\tau]\!] \;=\; \forall\beta.\,((\forall\alpha.\,([\![\tau]\!] \to \beta)) \to \beta) \qquad \text{if } \beta \mathrel{\#} \tau$$

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

$$\begin{array}{rcl} [\![\mathsf{pack}_{\exists\alpha.\tau}]\!] & : & \forall\alpha.\,([\![\tau]\!] \to [\![\exists\alpha.\tau]\!]) \\ & = & \Lambda\alpha.\lambda x{:}[\![\tau]\!].\,\Lambda\beta.\lambda k{:}\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x \\ [\![\mathsf{unpack}_{\exists\alpha.\tau}]\!] & : & [\![\exists\alpha.\tau]\!] \to \forall\beta.\,((\forall\alpha.\,([\![\tau]\!] \to \beta)) \to \beta) \\ & = & \lambda x{:}[\![\exists\alpha.\tau]\!].\,x \end{array}$$

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 constrants, their semantics is defined by seeing $\mathsf{pack}_{\exists\alpha.\tau}$ as a unary constructor and $\mathsf{unpack}_{\exists\alpha.\tau}$ as a unary destructor with the following reduction rule:

$$\mathsf{unpack}_{\exists\alpha.\tau_0}\,(\mathsf{pack}_{\exists\alpha.\tau}\;\tau'\;V) \;\longrightarrow\; \Lambda\beta.\lambda y{:}\forall\alpha.\,\tau \to \beta.\,y\;\tau'\;V \qquad (\delta_\exists)$$

Exercise 42 *Show that this δ-rule satisfies the progress and subject reduction assumptions for constants with the types in $\Delta_\exists$. (You may assume that the standard lemmas still hold.)* *(Solution p. 171)* □

Exercise 43 *The $\delta_\exists$ 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 proress for the δ-rule (Exercise 42)?* *(Solution p. 171)* □

Notice that our $\delta_\exists$-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 $\mathsf{unpack}_{\exists\alpha.\tau}$ as a binary destructor to avoid this intermediate step and have the more intuitive reduction rule:

$$\mathsf{unpack}_{\exists\alpha.\tau_0}\,(\mathsf{pack}_{\exists\alpha.\tau}\;\tau'\;V)\;\tau_1\;(\Lambda\alpha.\lambda x{:}\tau.\,M) \;\longrightarrow\; [x \mapsto V][\alpha \mapsto \tau']M \qquad (\delta_\exists)$$

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 δ-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:

$$V \;::=\; \ldots \mid \mathsf{pack}\;\tau', V\;\mathsf{as}\;\tau \qquad E \;::=\; \ldots \mid \mathsf{pack}\;\tau', [\,]\;\mathsf{as}\;\tau \mid \mathsf{let}\;\alpha, x = \mathsf{unpack}\;[\,]\;\mathsf{in}\;M$$

and by adding the following reduction rule:

$$\mathsf{let}\ \alpha, x = \mathsf{unpack}\ (\mathsf{pack}\ \tau', V\ \mathsf{as}\ \tau)\ \mathsf{in}\ M \longrightarrow [\alpha \mapsto \tau'][x \mapsto V]M$$

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, $\mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2$ cannot be reduced unless M_1 is itself a packed expression, which is indeed the case when M_1 is a value (or in head normal form). This contrasts with $\mathsf{let}\ x : \tau = M_1\ \mathsf{in}\ M_2$ where M_1 need not be evaluated and may be an application (*e.g.* in call-by-name or with strong reduction).

Exercise 45 *The reduction of* $\mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2$ *could be problematic when* M_1 *is not a value. Illustrate this on an example* (You may use the following hint if needed: *lanoitidnocaesu.)* *(Solution p. 172)* □

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 τ_0 of M is fully determined by M and the given type $\exists\alpha.\tau$ of the packed value. Checking that τ_0 is of the form $[\alpha \mapsto \tau']\tau$ 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-typed existential types

Intuitively, pack and unpack are just type information that can be dropped by type erasure. More precisely, the erasure of $\mathsf{pack}\ \tau', M\ \mathsf{as}\ \exists\alpha.\,\tau\exists\alpha.\,\tau$ is M and the erasure of $\mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2$ is a let-binding $\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2$. After type-erasure, the following typing rules for existential types in implicit-typed System F:

$$\textsc{if-Unpack}\quad \frac{\Gamma \vdash a_1 : \exists\alpha.\tau_1 \qquad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau_2} \qquad\qquad \textsc{if-Pack}\quad \frac{\Gamma \vdash a : [\alpha \mapsto \tau']\tau}{\Gamma \vdash a : \exists\alpha.\tau}$$

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

$$E ::= \ldots \mathsf{let}\ x = [\,]\ \mathsf{in}\ M \qquad\qquad \mathsf{let}\ x = V\ \mathsf{in}\ M \longrightarrow [x \mapsto V]M$$

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:

$$\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \longrightarrow [x \mapsto a_1]a_2$$

With existential types, this breaks subject reduction! This was first noticed by Sørensen and Urzyczyn (2006). See also (Fujita and Schubert, 2009, §9).

To see this, let τ_0 be $\exists\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$ and let v_0 be a value of type bool. Then, let v_1 and v_2 two values of type τ_0 with incompatible witness types, taking for instance, $\lambda f.\lambda x.1{+}(f\ (1+x))$ and $\lambda f.\lambda x.\mathit{not}\ (f\ (\mathit{not}\ x))$. Let v be the function $\lambda b.\,\text{if } b \text{ then } v_1 \text{ else } v_2$ of type $\mathsf{bool} \to \tau_0$, which returns either one of V_1 or V_2 depending on its argument b. We then have the reduction

$$a_1 \quad = \quad \mathsf{let}\ x = v\ v_0\ \mathsf{in}\ x\ (x\ (\lambda y.y)) \quad \longrightarrow \quad v\ v_0\ (v\ v_0\ (\lambda y.y)) \quad = \quad a_2$$

The typing judgment $\varnothing \vdash a_1 : \exists\alpha.\alpha \to \alpha$ holds, while $\varnothing \vdash a_2 : \tau$ does not hold for any τ. Indeed, the term a_1 is well-typed since $v\ v_0$ has type τ_0, hence x can be assumed of type $(\beta \to \beta) \to (\beta \to \beta)$ for some unknown type β 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 α. 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:

$$\frac{\Gamma \vdash a_1 : \exists\alpha.\tau_1 \qquad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash [x \mapsto a_1]a_2 : \tau_2}$$

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):

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

In the call-by-value setting, $\lambda k.[\![a]\!]\ k$ would come from the reduction of $[\![\mathsf{pack}\, a]\!]$, *i.e.* is $(\lambda k.\lambda x.k\ x)\ [\![a]\!]$, so that a is always a value v. However, a need not be a value. What is essential is again that a_1 be reduced to some head normal form $\lambda k.\, [\![a]\!]\; k$.

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 Läufer and Odersky (1994). Iso-existential types are explicitly *declared*, much as datatypes:

$$D\,\vec{\alpha} \approx \exists\bar{\beta}.\tau \qquad \text{if } \mathsf{ftv}(\tau) \subseteq \bar{\alpha} \cup \bar{\beta} \quad \text{and} \quad \bar{\alpha} \# \bar{\beta}$$

This introduces two constants, with the following type schemes:

$$\mathsf{pack}_D \;:\; \forall\bar{\alpha}\bar{\beta}.\,\tau \to D\,\vec{\alpha} \qquad\qquad \mathsf{unpack}_D \;:\; \forall\bar{\alpha}\gamma.\,D\,\vec{\alpha} \to (\forall\bar{\beta}.\,(\tau \to \gamma)) \to \gamma$$

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

Unfortunately, the "type scheme" of unpack_D is *not* an ML type scheme. A solution is to make unpack_D a binary primitive construct, rather than a constant, with an *ad hoc* typing rule:

$$\textsc{Unpack}_D\quad \frac{\Gamma \vdash M_1 : D\,\vec{\tau} \qquad \Gamma \vdash M_2 : \forall\bar{\beta}.\,([\vec{\alpha} \mapsto \vec{\tau}]\tau \to \tau_2) \qquad \bar{\beta} \# \vec{\tau}, \tau_2}{\Gamma \vdash \mathsf{unpack}_D\; M_1\; M_2 : \tau_2} \qquad \text{where } D\,\vec{\alpha} \approx \exists\bar{\beta}.\tau$$

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 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 not problematic. The construct unpack_D leads to this constraint generation rule (cf. §5):

$$\langle\!\langle \mathsf{unpack}_D\; M_1\; M_2 : \tau_2 \rangle\!\rangle \;=\; \exists\bar{\alpha}.\big(\; \langle\!\langle M_1 : D\,\vec{\alpha} \rangle\!\rangle \wedge \forall\bar{\beta}.\,\langle\!\langle M_2 : \tau \to \tau_2 \rangle\!\rangle \;\big)$$

where $D\,\vec{\alpha} \approx \exists\bar{\beta}.\tau$ and, *w.l.o.g.*, $\bar{\alpha}\bar{\beta} \# M_1, M_2, \tau_2$. Note that 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:

$$\mathsf{data}\; D\,\vec{\alpha} = \mathsf{forall}\; \bar{\beta}.\ell\;\tau$$

where ℓ is a data constructor. The elimination construct $\langle\!\langle \mathsf{case}\; M_1\; \mathsf{of}\; \ell\; x \to M_2 : \tau_2 \rangle\!\rangle$ and is typed as follows:

$$\langle\!\langle \mathsf{case}\; M_1\; \mathsf{of}\; \ell\; x \to M_2 : \tau_2 \rangle\!\rangle \;=\; \exists\bar{\alpha}.\big(\; \langle\!\langle M_1 : D\,\vec{\alpha} \rangle\!\rangle \wedge \forall\bar{\beta}.\,\mathsf{def}\; x : \tau\; \mathsf{in}\; \langle\!\langle M_2 : \tau_2 \rangle\!\rangle \;\big)$$

where, w.l.o.g., $\bar{\alpha}\bar{\beta} \# M_1, M_2, \tau_2$.

Examples Define $\mathsf{Any} \approx \exists\beta.\beta$. The following code that attempts to extract the raw content of a package fails:

$$\langle\!\langle \mathsf{unpack}_{\mathsf{Any}}\, M_1\ (\lambda x.\,x) : \tau_2 \rangle\!\rangle \;=\; \langle\!\langle M_1 : \mathsf{Any} \rangle\!\rangle \wedge \forall\beta.\,\langle\!\langle \lambda x.\,x : \beta \to \tau_2 \rangle\!\rangle \;\Vdash\; \forall\beta.\,\beta = \tau_2 \;\equiv\; \mathsf{false}$$

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

$$\begin{array}{rl} & \langle\!\langle \mathsf{unpack}_D\, M_1\ (\lambda(f,y).\,f\ y) : \tau \rangle\!\rangle \\ = & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \forall\beta.\,\langle\!\langle \lambda(f,y).\,f\ y : ((\beta \to \alpha) \times \beta) \to \tau \rangle\!\rangle) \\ \equiv & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \forall\beta.\,\mathsf{def}\ f : \beta \to \alpha; y : \beta\ \mathsf{in}\ \langle\!\langle f\ y : \tau \rangle\!\rangle) \\ \equiv & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \forall\beta.\,\tau = \alpha) \\ \equiv & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \tau = \alpha) \\ \equiv & \langle\!\langle M_1 : D\ \tau \rangle\!\rangle \end{array}$$

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).

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

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

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

The problem is that, in expression $\mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2$, occurrences of x can only be passed to polymorphic functions so that the type α 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:

$$\exists \mathsf{stack}.\{\ \mathit{empty} : \mathsf{stack};\ \ \mathit{push} : \mathsf{int} \times \mathsf{stack} \to \mathsf{stack};\ \ \mathit{pop} : \mathsf{stack} \to \mathsf{option}\ (\mathsf{int} \times \mathsf{stack})\ \}$$

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 appraoached described above.

For example, one may defined the previous datatype of frozen computations:

```
type 'a d = D : ('b → 'a) * 'b → 'a d
let freeze f x = D (f, x)
let run (D (f, x)) = f x
```

Here is the equivalent, more verbose code with modules:

```
module type D = sig type b 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 $[\![\tau_1 \to \tau_2]\!]$ must be $\exists\alpha.((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha$. Let us show that we may translate expressions so as to preserve well-typedness, *i.e.* so that $\Gamma \vdash M : \tau$ implies $[\![\Gamma]\!] \vdash [\![M]\!] : [\![\tau]\!]$. Assume $\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2$ and $\mathsf{dom}(\Gamma) = \{x_1, \ldots x_n\} = \mathsf{fv}(\lambda x{:}\tau_1.\, M)$. We may now hide the dependence on Γ using an existential type:

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

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

$$
\begin{array}{rcl}
[\![M\ M_1]\!] & = & \mathsf{let}\ \alpha, (\mathit{code} : (\alpha \times [\![\tau_1]\!]) \to \tau_2,\ \mathit{env} : \alpha) = \mathsf{unpack}\ [\![M]\!]\ \mathsf{in}\ \mathit{code}\ (\mathit{env}, [\![M_1]\!]) \\
& : & [\![\tau_2]\!]
\end{array}
$$

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

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

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

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

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

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

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

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

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

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

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

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

In summary, the environment-passing closure conversion is simple, but it requires the introduction of recursive non-functional values $\mathsf{let\ rec}\ \ x = V\ \mathsf{in}\ M$. 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).

6.3.2 Closure-passing closure conversion

Recall the *closure-passing* variant:

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

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

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:

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

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

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

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

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

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:* $\mathsf{proj}_i : \forall \alpha_1 \ldots \alpha_i \rho.\, \Pi\ (\alpha_1; \ldots \alpha_i; \rho) \to \alpha_i$.

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:

$$[\![\tau_1 \to \tau_2]\!] \;=\; \exists\rho.\mu\alpha.\Pi\;(((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!])\,;\rho)$$

ρ describes the environment represented as a row of fields, which is abstract; α is the concrete type of the closure that is to refer to recursively; $\Pi\;(((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!])\,;\rho)$ is a tuple that begins with a code pointer of type $(\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]$ and continues with the environment ρ. 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 ρ appears only once, which may seem surprising. Usually, an existential type variable appears both at positive and negative occurrences. Here, the variable α 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 $\mathsf{Clo}(R)$ abbreviate the concrete type of a closure of row R and $\mathsf{UClo}(R)$ its unfolded version:

$$\begin{array}{rcl}
\mathsf{Clo}(R) & \triangleq & \mu\alpha.\Pi\;((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R) \\
\mathsf{UClo}(R) & \triangleq & \Pi\;((\mathsf{Clo}(R) \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R)
\end{array}$$

The encoding of arrow types $[\![\tau_1 \to \tau_2]\!]$ is $\exists\rho.\mathsf{Clo}(\rho)$. The encoding of abstactions and applications is:

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

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

where $\{x_1, \ldots, x_n\} = \mathsf{fv}(\lambda x\!:\!\tau_1.\,M)$.

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

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

where $\{x_1, \ldots, x_n\} = \mathsf{fv}(\mu f.\lambda x.a)$. 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.* □

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 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 F.

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

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

Remind that $\mathsf{Clo}(R)$ abbreviates $\mu\alpha.\Pi\ ((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R)$. Hence, $\vec{\beta}$ are free variables of $\mathsf{Clo}(R)$. 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, their is some flexibility in the representation of the closure. For instance, the following change is completely local:

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

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 $\mu(f_1, f_2).(\lambda x_1.\, a_1, \lambda x_2.\, a_2)$, say a?

The environment passing encoding is as follows:

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

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

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

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:

$$
\begin{array}{rcl}
[\![a]\!] & = & \text{let } code_1 = \lambda(clo, x).\, \text{let } (\_, \_, f_1, f_2, x_1, \ldots, x_n) = clo \text{ in } [\![a_1]\!] \text{ in} \\
& & \text{let } code_2 = \lambda(clo, x).\, \text{let } (\_, f_1, f_2, x_1, \ldots, x_n) = clo \text{ in } [\![a_2]\!] \text{ in} \\
& & \text{let rec } clo_1 = (code_1, code_2, clo_1, clo_2, x_1, \ldots, x_n) \\
& & \quad \text{and } clo_2 = clo_1.tail \\
& & \ \text{in } clo_1, clo_2
\end{array}
$$

Here $clo_1.tail$ returns a pointer to the tail $(code_2, clo_1, clo_2, x_1, \ldots, x_n)$ of clo_1 without allocating a new tuple.

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:

$$\text{class } c\ (x_1, \ldots x_q)\ \{\text{meth } \ m_1 = a_i; \ldots \text{meth } \ m_q = a_i\}$$

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:

$$
\begin{array}{l}
\text{let } m = \\
\qquad \{ \quad m_1 = \lambda(m, x_1, \ldots x_q).\, [\![a_1]\!]; \\
\qquad\qquad \vdots \\
\qquad\quad\ \ m_p = \lambda(m, x_1, \ldots x_q).\, [\![a_p]\!] \quad \} \text{ in} \\
\lambda x_1, \ldots x_q.\, (m, x_1, \ldots x_q)
\end{array}
$$

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

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

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

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

Calling a method of an object p of type M is

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

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

Typed encoding of objects 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^ω: 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:*

$$\begin{aligned} [\![V]\!] &= \lambda k.\, k\, (\!|V|\!) \\ [\![M_1\ M_2]\!] &= \lambda k.\, [\![M_1]\!]\ (\lambda x_1.\, [\![M_2]\!]\ (\lambda x_2.\, x_1\ x_2\ k)) \end{aligned}$$

$$\begin{aligned} (\!|x|\!) &= x \\ (\!|()|\!) &= () \\ (\!|(V_1, V_2)|\!) &= ((\!|V_1|\!), (\!|V_2|\!)) \\ (\!|\lambda x.\, M|\!) &= \lambda x.\, [\![M]\!] \end{aligned}$$

Is this a type-preserving transformation? (Solution p. 172) □

Appendix A

Proofs and Answers to exercises

Solution of Exercise 42

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

$$\Gamma \vdash \mathsf{unpack}_{\exists\alpha.\tau_1}\,(\mathsf{pack}_{\exists\alpha.\tau}\,\tau'\,V) : \tau_0$$

By inversion of typing, τ_1 and τ_0 must be equal to τ and $\forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$, respectively, and the judgment $\Gamma \vdash V : [\alpha \mapsto \tau']\tau$ must hold. Let Γ' be $\Gamma, \beta, y : \forall\alpha.\,\tau \to \beta$. By weakening, we have $\Gamma' \vdash V : [\alpha \mapsto \tau']\tau$. We then have $\Gamma' \vdash y\,\tau'\,V : \beta$ and finally, we have

$$\Gamma \vdash \Lambda\beta.\lambda y{:}\forall\alpha.\,\tau \to \beta.\,y\,\tau'\,V : \tau_0$$

as expected.

We then need to show that $\delta_\exists$ satisfies progress, *i.e.*, a full well typed application of $\mathsf{unpack}_{\exists\alpha.\tau}$ can always be reduced. Assume that $\Gamma \vdash \mathsf{unpack}_{\exists\alpha.\tau}\,V : \tau_0$. By inversion of typing, we must have $\Gamma \vdash V : \exists\alpha.\,\tau$. By the classification lemma (to be extended and rechecked), V must be an existential value, *i.e.* of the form $\mathsf{pack}_{\exists\alpha.\tau_1}\,\tau_0\,V_0$. Hence, $\mathsf{unpack}_{\exists\alpha.\tau}\,V$ reduces by $\delta_\exists$. ■

Solution of Exercise 43

We just force τ_1 to coincide with τ:

$$\mathsf{unpack}_{\exists\alpha.\tau}\,(\mathsf{pack}_{\exists\alpha.\tau}\,\tau'\,V) \;\longrightarrow\; \Lambda\beta.\lambda y{:}\forall\alpha.\,\tau \to \beta.\,y\,\tau'\,V \qquad (\delta_\exists)$$

The proof of subject reduction will know by construction that τ_0 is τ instead of learning it by inversion of typing. Conversely for progress, we will have to show that τ_1 and τ are equal by inversion so that $\delta_\exists$ can be applied. ■

Solution of Exercise 45

Let M_1 be $\mathsf{if}\ M\ \mathsf{then}\ V_1\ \mathsf{else}\ V_2$ where V_i is of the form $\mathsf{pack}\ \tau_i, V_i\ \mathsf{as}\ \exists\alpha\tau$ and the two witnesses τ_1 and τ_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.

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

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

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

▷ Lennart Augustsson. Implementing Haskell overloading. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 65–73, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X.

▷ Nick Benton and Andrew Kennedy. Exceptional syntax journal of functional programming. *J. Funct. Program.*, 11(4):395–410, 2001.

▷ Richard Bird and Lambert Meertens. Nested datatypes. In *International Conference on Mathematics of Program Construction (MPC)*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

Nikolaj Skallerud Bjørner. Minimal typing derivations. In *In ACM SIGPLAN Workshop on ML and its Applications*, pages 120–126, 1994.

Daniel Bonniot. *Typage modulaire des multi-méthodes*. PhD thesis, École des Mines de Paris, November 2005.

▷ Daniel Bonniot. Type-checking multi-methods in ML (a modular approach). In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 2002.

▷ Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33:309–338, 1998.

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

Luca Cardelli. An implementation of f¡:. Technical report, DEC Systems Research Center, 1993.

Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.

▷ Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. The MLton compiler, 2007.

▷ Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

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

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

▷ Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

Julien Crétin and Didier Rémy. Extending System F with Abstraction over Erasable Coercions. In *Proceedings of the 39th ACM Conference on Principles of Programming Languages*, January 2012.

Joshua Dunfield. Greedy bidirectional polymorphism. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 15–26, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-509-3. doi: http://doi.acm.org/10.1145/1596627.1596631.

▷ Ken-etsu Fujita and Aleksy Schubert. Existential type systems with no types in terms. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, pages 112–126, 2009. doi: 10.1007/978-3-642-02273-9_10.

Jun Furuse. Extensional polymorphism by flow graph dispatching. In Ohori (2003), pages 376–393. ISBN 3-540-20536-5.

▷ Jun Furuse. Extensional polymorphism by flow graph dispatching. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science*. Springer, November 2003b.

▷ Jacques Garrigue. Relaxing the value restriction. In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, April 2004.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université Paris 7, June 1972.

▷ Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1990.

▷ Dan Grossman. Quantified types in an imperative language. *ACM Transactions on Programming Languages and Systems*, 28(3):429–475, May 2006.

▷ Bob Harper and Mark Lillibridge. ML with callcc is unsound. Message to the TYPES mailing list, July 1991.

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

▷ Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

▷ J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

▷ Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *ACM SIGPLAN Conference on History of Programming Languages*, June 2007.

Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris 7, September 1976.

▷ John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

▷ Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 160–169, New York, NY, USA, 1995a. ACM. ISBN 0-89791-719-7.

Mark P. Jones. Typing Haskell in Haskell. In *In Haskell Workshop*, 1999a.

Mark P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995b. ISBN 0-521-47253-9.

▷ Mark P. Jones. Typing Haskell in Haskell. In *Haskell workshop*, October 1999b.

▷ Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, 1997.

▷ Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(01):1, 2006.

Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 193–204, New York, NY, USA, 1992. ACM. ISBN 0-89791-481-3. doi: http://doi.acm.org/10.1145/141471.141540.

▷ Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, pages 206–220. Springer, May 1990.

▷ Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Communications of the ACM*, 8(2):89–101, 1965.

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

▷ Didier Le Botlan and Didier Rémy. Recasting MLF. *Information and Computation*, 207(6): 726–785, 2009. ISSN 0890-5401. doi: 10.1016/j.ic.2008.12.006.

▷ Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris 7, June 1992.

▷ Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54, January 2006.

▷ Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/349214.349230.

▷ John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–57, January 1988.

▷ Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 382–401, 1990.

▷ David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer, June 2003.

Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 279–303, London, UK, 1999. Springer-Verlag. ISBN 3-540-66156-5.

▷ Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

▷ Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–283, January 1996.

▷ John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2–3):211–249, 1988.

▷ John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

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

J. Garrett Morris and Mark P. Jones. Instance chains: type class programming without overlapping instances. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 375–386, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: http://doi.acm.org/10.1145/1863543.1863596.

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

▷ Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

▷ Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, April 1984.

▷ Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. Functional logic overloading. pages 233–244, 2002. doi: http://doi.acm.org/10.1145/565816.503294.

▷ Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 135–146, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7.

▷ Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

▷ Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 41–53, 2001.

Atsushi Ohori, editor. *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings*, volume 2895 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-20536-5.

▷ Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

▷ Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: a new foundation for generic programming. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 35–44, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254070.

▷ Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Online lecture notes, January 2009.

▷ Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Manuscript, April 2004.

▷ Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, January 1993.

Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: http://doi.acm.org/10.1145/62678.62697.

▷ Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

▷ Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

▷ Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

▷ François Pottier. Notes du cours de DEA "Typage et Programmation", December 2002.

François Pottier. A typed store-passing translation for general references. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, Austin, Texas, January 2011. Supplementary material.

François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of Functional Programming*, 23(1):38–144, January 2013.

François Pottier. Hindley-Milner elaboration in applicative style. In *Proceedings of the 2014 ACM SIGPLAN International Conference on Functional Programming (ICFP'14)*, September 2014.

▷ François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. Submitted for publication, October 2012.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 173–184, September 2013.

▷ François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

▷ François Pottier and Didier Rémy. The essence of ML type inference. Draft of an extended version. Unpublished, September 2003.

▷ Didier Rémy. Simple, partial type-inference for System F based on type-containment. In *Proceedings of the tenth International Conference on Functional Programming*, September 2005.

▷ Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346. Springer, April 1994a.

▷ Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming: Types, Semantics and Language Design*. MIT Press, 1994b.

▷ Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

Didier Rémy and Boris Yakobowski. Efficient Type Inference for the MLF language: a graphical and constraints-based approach. In *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 63–74, Victoria, BC, Canada, September 2008. doi: http://doi.acm.org/10.1145/1411203.1411216.

▷ John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, April 1974.

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

▷ John C. Reynolds. Three approaches to type structure. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer, March 1985.

François Rouaix. Safe run-time overloading. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 355–366, 1990. doi: http://doi.acm.org/10.1145/96709.96746.

▷ Christian Skalka and François Pottier. Syntactic type soundness for HM(X). In *Workshop on Types in Programming (TIP)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, July 2002.

Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. In *Science of Computer Programming*, 1994.

Morten Heine Sørensen and Pawel Urzyczyn. *Studies in Logic and the Foundations of Mathematics*, chapter Lectures on the Curry-Howard Isomorphism. Elselvir Science Inc, 2006.

▷ Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In Sofiène Tahar, Otmame Ait-Mohamed, and César Muñoz, editors, *TPHOLs 2008: Theorem Proving in Higher Order Logics, 21th International Conference*, Lecture Notes in Computer Science. Springer, August 2008.

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

▷ Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000.

▷ Peter J. Stuckey and Martin Sulzmann. A theory of overloading. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 167–178, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8.

▷ Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 11(2):245–296, 1994.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

▷ Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179(1):1–18, 2002.

▷ Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, September 2004.

▷ Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

▷ Philip Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, September 1989.

▷ Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1–3):201–226, May 2007.

▷ Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–76, January 1989.

Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

▷ J. B. Wells. The essence of principal typings. In *International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer, 2002.

▷ J. B. Wells. The undecidability of Mitchell's subtyping relation. Technical Report 95-019, Computer Science Department, Boston University, December 1995.

▷ J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

▷ Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8 (4):343–356, December 1995.

▷ Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.