MPRI 2.4, Functional programming and type systems
Metatheory of System F

Didier Rémy

Plan of the course

Metatheory of System F

ADTs, Recursive types, Existential types, GATDs

Going higher order with F^ω!

Logical relations

Side effects, References, Value restriction

Type reconstruction

Overloading

Abstract Data types, Existential types, GADTs

Contents

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Algebraic Datatypes Types Examples

In OCaml:

```
type 'a list =
  | Nil  : 'a list
  | Cons : 'a * 'a list → 'a list
```

or

```
type ('leaf, 'node) tree =
  | Leaf : 'leaf → ('leaf, 'node) tree
  | Node : ('leaf, 'node) tree * 'node * ('leaf, 'node) tree → ('leaf, 'node) tree
```

Algebraic Datatypes Types General case

General case

$$\text{type } G\,\vec{\alpha} = \Sigma_{i\in 1..n}(C_i : \forall\vec{\alpha}.\,\tau_i \to G\,\vec{\alpha}) \qquad \text{where } \vec{\alpha} = \bigcup_{i\in 1..n}\mathrm{ftv}(\tau_i)$$

In System F, this amounts to declaring:

- a new type constructor G,
- n constructors $C_i \;:\; \forall\vec{\alpha}.\,\tau_i \to G\,\vec{\alpha}$
- one destructor $d_G \;:\; \forall\vec{\alpha},\gamma.\,G\,\vec{\alpha} \to (\tau_1 \to \gamma)\ldots(\tau_n \to \gamma) \to \gamma$
- n reduction rules $d_G\,\bar{\tau}\;(C_i\;\bar{\tau}'\;v)\;v_1\;\ldots v_n \;\rightsquigarrow\; v_i\;v$

Exercise

Show that this extension verifies the subject reduction and progress axioms for constants.

Algebraic Datatypes Types

General case

$$\text{type } G\,\vec{\alpha} = \Sigma_{i\in 1..n}(C_i : \forall\vec{\alpha}.\,\tau_i \to G\,\vec{\alpha}) \qquad \text{where } \vec{\alpha} = \bigcup_{i\in 1..n}\mathrm{ftv}(\tau_i)$$

Notice that

- All constructors build values of the same type $G\ \vec{\alpha}$ and are surjective (all types can be reached)
- The definition may be recursive, *i.e.* G may appear in τ_i

Algebraic datatypes introduce *isorecursive types*.

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Recursive Types

Product and sum types alone do not allow describing *data structures* of *unbounded size*, such as lists and trees.

Indeed, if the grammar of types is $\tau ::= \mathit{unit} \mid \tau \times \tau \mid \tau + \tau$, then it is clear that every type describes a *finite* set of values.

For every k, the type of lists of length at most k is expressible using this grammar. However, the type of lists of unbounded length is not.

Equi- versus isorecursive types

The following definition is inherently *recursive:*

"A list is either empty or a pair of an element and a list."

We need something like this:

$$\textit{list}\ \alpha \quad \diamond \quad \textit{unit} + \alpha \times \textit{list}\ \alpha$$

But what does $\diamond$ stand for? Is it *equality,* or some kind of *isomorphism?*

There are two standard approaches to recursive types:

- *equirecursive* approach:
  a recursive type is *equal* to its unfolding.
- *isorecursive* approach:
  a recursive type and its unfolding are related via explicit *coercions*.

Equirecursive types

In the equirecursive approach, the usual syntax of types:

$$\tau ::= \alpha \mid \mathsf{F}\,\vec{\tau} \mid \forall\beta.\tau$$

is no longer interpreted inductively. Instead, types are the *regular infinite trees* built on top of this grammar.

Finite syntax for recursive types

$$\tau ::= \alpha \mid \mu\alpha.(\mathsf{F}\,\vec{\tau}) \mid \mu\alpha.(\forall\beta.\tau)$$

We do not allow the seemingly more general form $\mu\alpha.\tau$, because $\mu\alpha.\alpha$ is meaningless, and $\mu\alpha.\beta$ or $\mu\alpha.\mu\beta.\tau$ are useless. If we write $\mu\alpha.\tau$, it should be understood that τ is contractive, that is, τ is a type constructor application or a forall introduction.

For instance, the type of lists of elements of type α is:

$$\mu\beta.(\mathit{unit} + \alpha \times \beta)$$

Equirecursive types

Inductive definition [Brandt and Henglein, 1998] show that equality is the least congruence generated by the following two rules:

Fold/Unfold
$$\mu\alpha.\tau = [\alpha \mapsto \mu\alpha.\tau]\tau$$

Uniqueness
$$\frac{\tau_1 = [\alpha \mapsto \tau_1]\tau \qquad \tau_2 = [\alpha \mapsto \tau_2]\tau}{\tau_1 = \tau_2}$$

In both rules, τ must be contractive.

This axiomatization does not directly lead to an efficient algorithm for deciding equality, though.

Co-inductive definition

$$\alpha = \alpha \qquad \frac{[\alpha \mapsto \mu\alpha.\mathsf{F}\vec{\tau}]\vec{\tau} = [\alpha \mapsto \mu\alpha.\mathsf{F}\vec{\tau}']\vec{\tau}'}{\mu\alpha.\mathsf{F}\vec{\tau} = \mu\alpha.\mathsf{F}\vec{\tau}'} \qquad \frac{[\alpha \mapsto \mu\alpha.\forall\beta.\tau]\tau = [\alpha \mapsto \mu\alpha.\forall\beta.\tau']\tau'}{\mu\alpha.\forall\beta.\tau = \mu\alpha.\forall\beta.\tau'}$$

Equirecursive types

Equality

In the absence of quantifiers

Each type in this syntax denotes a unique regular tree, sometimes known as its *infinite unfolding.* Conversely, every regular tree can be expressed in this notation (possibly in more than one way).

If one builds a type-checker on top of this finite syntax, then one must be able to *decide* whether two types are *equal,* that is, have identical infinite unfoldings.

This can be done efficiently, either via the algorithm for comparing two DFAs, or better, by unification. (The latter approach is simpler, faster, and extends to the type inference problem.)

Exercise

Show that $\mu\alpha.A\alpha = \mu\alpha.AA\alpha$ *and* $\mu\alpha.AB\alpha = A\mu\alpha.BA\alpha$ *with both inductive and co-inductive definitions. Can you do it without the* UNIQUENESS *rule?*

Equirecursive types

Without quantifiers

Proof of $\mu\alpha A A\alpha = \mu\alpha A A A\alpha$

By coinduction

Let | u be $\mu\alpha A A\alpha$
 | v be $\mu\alpha A A A\alpha$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\mathbf{(1)}}{Au = Av}}{u = AAv}}{Au = v}}{u = Av}}{Au = AAv}}{u = v \quad (1)}$$

By unification

Equivalent classes, using *small terms*	To do:
$u \sim Au_1 \quad \wedge \quad u_1 \sim Au \quad \wedge \quad v \sim Av_1 \quad \wedge \quad v_1 \sim Av_2 \quad \wedge \quad v_2 \sim Av$	$u \sim v$
$u \sim Au_1 \sim v \sim Av_1 \quad \wedge \quad u_1 \sim Au \quad \wedge \quad v_1 \sim Av_2 \quad \wedge \quad v_2 \sim Av$	$u_1 \sim v_1$
$u \sim v \sim Av_1 \quad \wedge \quad u_1 \sim Au \sim v_1 \sim Av_2 \quad \wedge \quad v_2 \sim Av$	$u \sim v_2$

Equirecursive types

In the presence of quantifiers

The situation is more subtle because of α-conversion.

A (somewhat involved) canonical form can still be found, so that checking equality and first-order unification on types can still be done in $O(n \log n)$. See [Gauthier and Pottier, 2004].

Otherwise, without the use of such canonical forms, the best known algorithm is in $O(n^2)$ [Glew, 2002] testing equality of automatons with binders.

Equirecursive types — With quantifiers

Example of unfolding with canonical forms [Gauthier and Pottier, 2004].

- the letter in green, is just any name, subject to α-conversion
- the number is the canonical name: it is the number of free variables under the binder—including recursive occurrences.

$$\begin{aligned}
& \forall a1.\, \mu\ell.a1 \to \forall a2.\,(a2 \to \ell) && (1)\\
& \forall a1.\, \mu\ell.a1 \to \forall b2.\,(b2 \to \ell) && (\alpha)\\
= {} & \forall a1.\quad a1 \to \forall b2.\,(b2 \to \mu\ell.a1 \to \forall b2.\,(b2 \to \ell)) && (\mu)\\
= {} & \forall a1.\quad a1 \to \forall b2.\,(b2 \to \mu\ell.a1 \to \forall c2.\,(c2 \to \ell)) && (\alpha)
\end{aligned}$$

With the canonical representation,

- Syntactic unfolding (*i.e.* without any renaming) avoids name capture and is also a correct semantical unfolding
- It shares free variables and can reuse the same name for the new bound variables without name capture.

Equirecursive types

In the presence of equirecursive types, structural induction on types is no longer permitted, but *we never used it* anyway – in soundness proofs.

We only need it to prove the termination of reduction, which does not hold any longer.

It remains true that

- $\mathsf{F}\ \vec{\tau}_1 = \mathsf{F}\ \vec{\tau}_2$ implies $\vec{\tau}_1 = \vec{\tau}_2$ (symbols are injective)—this is used in the proof of Subject Reduction.
- $\mathsf{F}_1\ \vec{\tau}_1 = \mathsf{F}_2\ \vec{\tau}_2$ implies $\mathsf{F}_1 = \mathsf{F}_2$—this is used in the proof of Progress.

So, the reasoning that leads to *type soundness* is unaffected.

Exercise

Prove type soundness for the simply-typed λ-calculus in Coq. Then, change the syntax of types from `Inductive` *to* `CoInductive`*.*

Equirecursive types break termination, indeed!

That is no a surprise, but...

What is the expressiveness of simply-typed λ-calculus with equirecursive types alone (no other constructs and/or constants)?

All terms of the untyped λ-calculus are typable!

- define the universal type U as $\mu\alpha.\alpha \to \alpha$
- we have $U = U \to U$, hence all terms are typable with type U.

Notce that one can emulate recursive types $U = U \to U$ by defining two functions *fold* and *unfold* of respective types $(U \to U) \to U$ and $U \to (U \to U)$ with side effects, such as:

- references, or
- exceptions

Equirecursive types in OCaml

OCaml has both isorecursive and equirecursive types.

- equirecursive types are restricted by default to objects or datatypes.
- unrestricted equirecursive types are available upon explicit request.

Quiz: why so?

Isorecursive types

The folding/unfolding is witnessed by an explicit coercion.

The uniqueness rule is often omitted

(hence, the equality relation is weaker)

Encoding isorecursive types with ADT

The recursive type $\mu\beta.\tau$ can be represented in System F by introducing a datatype with a unique constructor:

$$\text{type } G\,\vec{\alpha} = \Sigma(C : \forall\vec{\alpha}.\,[\beta \mapsto G\,\vec{\alpha}]\tau \to G\,\vec{\alpha}) \qquad \text{where } \vec{\alpha} = \mathrm{ftv}(\tau) \setminus \{\beta\}$$

For any $\vec{\alpha}$, the constructor $C\vec{\alpha}$ coerces $[\beta \mapsto G\,\vec{\alpha}]\tau$ to $G\,\vec{\alpha}$ and the reverse coercion is the function $\lambda x{:}G\,\vec{\alpha}.\,d_G\,\vec{\alpha}\;x\;(\lambda y.\,y)$.

Since this datatype has a unique constructor, pattern matching always succeeds and amounts to the identity. Hence, in $\lceil F \rceil$, the constructor could be removed: coercions have no computational content.

Records

A record can be defined as

$$\text{type } G\,\vec{\alpha} = \Pi_{i\in 1..n}(\ell_i : \tau_i) \qquad \text{where } \vec{\alpha} = \bigcup_{i\in 1..n} \mathrm{ftv}(\tau_i)$$

Exercise

What are the corresponding declarations in System F?

- *a new type constructor* G_Π,
- 1 *constructor* $\quad C_\Pi \;:\; \forall\vec{\alpha}.\,\tau_1 \to \ldots \tau_n \to G\,\vec{\alpha}$
- n *destructors* $\quad d_{\ell_i} \;:\; \forall\vec{\alpha}.\,G\,\vec{\alpha} \to \tau_i$
- n *reduction rules* $\quad d_{\ell_i}\,\bar{\tau}\,(C_\Pi\,\bar{\tau}\,v_1\,\ldots v_n) \;\rightsquigarrow\; v_i$

Can a record also be used for defining recursive types?

Exercise

Show type soundness for records.

Deep pattern matching

In practice, one allows deep pattern matching and wildcards in patterns.

```
type nat = Z | S of nat
let rec equal n1 n2 = match n1, n2 with
  | Z, Z → true
  | S m1, S m2 → equal m1 m2
  | _ → false
```

Then, one should check for *exhaustiveness* of pattern matching.

Deep pattern matching can be compiled away into shallow patterns—or directly compiled to efficient code.

See [Le Fessant and Maranget, 2001; Maranget, 2007]

Exercise

Do the transofrmation manually for the function equal.

ADTs Regular

$$\text{type } G\ \vec{\alpha} = \Sigma_{i\in 1..n}(C_i : \forall\vec{\alpha}.\, \tau_i \to G\ \vec{\alpha})$$

If all occurrences of G in τ_i are $G\ \vec{\alpha}$ then, the ADT is *regular*.

Remark regular ADTs can be encoded in System-F. (More precisely, the church encodings of regular ADTs are typable in System-F.)

ADTs Non Regular

Non-regular ADT's do not have this restriction:

```
type 'a seq =
  | Nil
  | Zero of ('a * 'a) seq
  | One of 'a * ('a * 'a) seq
```

They usually need *polymorphic* recursion to be manipulated.

Non regular ADT are heavily used by Okasaki [1999] for implementing purely functional data structures.

(They are also typically used with GADTs.)

Non-regular ADT can actually be encoded in F^ω.

Contents

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Existential types *Examples*

A frozen application returning a value of type (≈ a thunk)

$$\exists\alpha.(\alpha \to \tau) \times \alpha$$

Type of closures in the environment-passing variant:

$$[\![\tau_1 \to \tau_2]\!] \quad = \quad \exists\alpha.((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha$$

A possible encoding of objects:

$$\begin{array}{lll}
= & \exists\rho. & \rho \text{ describes the state} \\
 & \quad \mu\alpha. & \alpha \text{ is the concrete type of the closure} \\
 & \qquad \Pi\ (& \text{a tuple...} \\
 & \qquad\quad \{(\alpha \times \tau_1) \to \tau_1'; & \text{... that begins with a record...} \\
 & \qquad\qquad \ldots & \\
 & \qquad\qquad (\alpha \times \tau_n) \to \tau_n'\ \}\ ; & \text{... of method code pointers...} \\
 & \qquad\quad \rho & \text{...and continues with the state} \\
 & \qquad) & \quad \text{(a tuple of unknown length)}
\end{array}$$

Existential types

Let's first look at the type-erasing interpretation, with an explicit notation for introducing and eliminating existential types.

Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

$$\textsc{Pack}\quad \frac{\Gamma \vdash M : [\alpha \mapsto \tau']\tau}{\Gamma \vdash \textit{pack}\ \tau', M\ \textit{as}\ \exists\alpha.\,\tau : \exists\alpha.\,\tau}$$

$$\textsc{Unpack}\quad \frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma, \alpha, x : \tau_1 \vdash M_2 : \tau_2 \qquad \alpha \mathrel{\#} \tau_2}{\Gamma \vdash \textit{let}\ \alpha, x = \textit{unpack}\ M_1\ \textit{in}\ M_2 : \tau_2}$$

Anything wrong?The side condition $\alpha \mathrel{\#} \tau_2$ is mandatory here to ensure well-formedness of the conclusion.

The side condition may also be written $\Gamma \vdash \tau_2$ which implies $\alpha \mathrel{\#} \tau_2$, given that the well-formedness of the last premise implies $\alpha \notin \mathrm{dom}(\Gamma)$.

Note the *imperfect* duality between universals and existentials:

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

On existential elimination

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

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

Informally, this could mean that, if M has type τ for some *unknown* α, then it has type τ, where α is “fresh”...

Why is this broken?

We could immediately *universally* quantify over α, and conclude that $\Gamma \vdash \Lambda\alpha.\, \textit{unpack}\, M : \forall\alpha.\,\tau$. This is nonsense!

Replacing the premise $\Gamma, \alpha \vdash M : \exists\alpha.\tau$ by the conjunction $\Gamma \vdash M : \exists\alpha.\tau$ and $\alpha \in \mathrm{dom}(\Gamma)$ would make the rule even more permissive, so it wouldn't help.

On existential elimination

A correct elimination rule must force the existential package to be *used* in a way that does not rely on the value of α.

Hence, the elimination rule must have control over the *user* of the package—that is, over the term M_2.

$$\textsc{Unpack}\quad \frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma, \alpha; x : \tau_1 \vdash M_2 : \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \textit{let}\ \alpha, x = \textit{unpack}\ M_1\ \textit{in}\ M_2 : \tau_2}$$

The restriction $\alpha \# \tau_2$ prevents writing "*let* α, x = *unpack* M_1 *in* x", which would be equivalent to the unsound "*unpack* M" of the previous slide.

The fact that α is bound within M_2 forces it to be treated abstractly.

In fact, M_2 must be ??? in α.

On existential elimination

In fact, M_2 must be *polymorphic* in α: the second premise could be:

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

or, if N_2 stands for $\Lambda\alpha.\,\lambda x{:}\tau_1.\,M_2$:

$$\frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma \vdash N_2 : \forall\alpha.\,\tau_1 \to \tau_2 \qquad \alpha \mathrel{\#} \tau_2}{\Gamma \vdash \textit{unpack}\ M_1\ N_2 : \tau_2}$$

One could even view "$\textit{unpack}_{\exists\alpha.\tau_1}$" as a family of *constants* of types:

$$\textit{unpack}_{\exists\alpha.\tau_1} : \quad (\exists\alpha.\tau_1) \to \big(\forall\alpha.\,(\tau_1 \to \tau_2)\big) \to \tau_2 \qquad \alpha \mathrel{\#} \tau_2$$

Thus, $\textit{unpack}_{\exists\alpha.\tau} : \quad \forall\beta.\big((\exists\alpha.\tau) \to (\forall\alpha.\,(\tau \to \beta)) \to \beta\big)$

or, better $\textit{unpack}_{\exists\alpha.\tau} : \quad (\exists\alpha.\tau) \to \forall\beta.\big((\forall\alpha.\,(\tau \to \beta)) \to \beta\big)$

β stands for τ_2: it is bound prior to α, so it cannot be instantiated to a type that refers to α, which reflects the side condition $\alpha \mathrel{\#} \tau_2$.

On existential introduction

$$\textsc{Pack}\quad \frac{\Gamma \vdash M : [\alpha \mapsto \tau']\tau}{\Gamma \vdash \textit{pack}\ \tau', M\ \textit{as}\ \exists\alpha.\,\tau : \exists\alpha.\tau}$$

Hence, "$\textit{pack}_{\exists\alpha.\tau}$" can be viewed as a family *constant* of types:

$$\textit{pack}_{\exists\alpha.\tau} : \quad [\alpha \mapsto \tau']\tau \to \exists\alpha.\tau$$

i.e. of polymorphic types:

$$\textit{pack}_{\exists\alpha.\tau} : \quad \forall\alpha.\,(\tau \to \exists\alpha.\tau)$$

Existentials as constants

In System F, existential types can be presented as a family of constants:

$$\begin{array}{rcl} \mathit{pack}_{\exists\alpha.\tau} & : & \forall\alpha.(\tau \to \exists\alpha.\tau) \\ \mathit{unpack}_{\exists\alpha.\tau} & : & \exists\alpha.\tau \to \forall\beta.((\forall\alpha.(\tau \to \beta)) \to \beta) \end{array}$$

Read:

- for *any* α, if you have a τ, then, for *some* α, you have a τ;
- if, for *some* α, you have a τ, then, (for any β,) if you wish to obtain a β out of it, 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, $\neg(\forall x.\neg F)$.

Is there an encoding of existential types into universal types?

Encoding existentials into universals

The type translation is *double negation:*

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

The term translation is:

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

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

What is the computational content of this encoding?

A *continuation-passing transform.*

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

The semantics of existential types as constants

pack$_{\exists\alpha.\tau}$ can be treated as a unary constructor, and *unpack*$_{\exists\alpha.\tau}$ as a unary destructor. The δ-reduction rule is:

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

It would be more intuitive, however, to treat *unpack*$_{\exists\alpha.\tau_0}$ as a binary destructor:

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

Remark:

- This does not quite fit in our generic framework for constants, which must receive all type arguments prior to value arguments.
- But our framework could be easily extended.

The semantics of existential types as primitive

We extend values and evaluation contexts as follows:

$$\begin{array}{lcl} V & ::= & \ldots \mid \mathit{pack}\,\tau', V\ \mathit{as}\,\tau \\ E & ::= & \ldots \mid \mathit{pack}\,\tau', []\ \mathit{as}\,\tau \mid \mathit{let}\,\alpha, x = \mathit{unpack}\,[]\ \mathit{in}\,M \end{array}$$

We add the reduction rule:

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

Exercise

Show that subject reduction and progress hold.

The semantics of existential types beware!

The reduction rule for existentials destructs its arguments.

Hence, *let* α, x = *unpack* M_1 *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 *let* $x : \tau$ = M_1 *in* M_2 where M_1 need not be evaluated and may be an application (*e.g.* with call-by-name or strong reduction strategies).

The semantics of existential types beware!

Exercise

Find an example that illustrates why the reduction of
let α, x = *unpack* M_1 *in* M_2 *could be problematic when* M_1 *is not a value.*

Need a hint?

Use a conditional *Solution*

Let M_1 be if M then V_1 else V_2 where V_i is of the form
pack τ_i, W_i *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.

Is pack too verbose?

Exercise

Recall the typing rule for pack:

$$\frac{\Gamma \vdash M : [\alpha \mapsto \tau']\tau}{\Gamma \vdash \textit{pack}\, \tau', M \;\textit{as}\; \exists\alpha.\,\tau : \exists\alpha.\tau}$$

Isn't the witness type τ' annotation superfluous?

- The type τ_0 of M is fully determined by M. Given the 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.

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Implicitly-typed existential types

Intuitively, pack and unpack are just type annotations that could be dropped, leaving a let-binding instead of the unpack form.

Hence, the typing rule for implicitly-typed existential types:

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

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

The semantics of this let-binding is as before:

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

Is the semantics type-erasing?

Implicitly-typed existential types subtlety

Yes, it is.

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

We chose a call-by-value semantics, but so far, as long as there is no side-effect, we could have chosen a call-by-name semantics (or even perform reduction under abstraction).

In a call-by-name semantics, the let-bound expression is not reduced prior to substitution in the body:

$$\textit{let } x = M_1 \textit{ in } M_2 \longrightarrow [x \mapsto M_1]M_2$$

With existential types, this breaks subject reduction!

Why?

Implicitly-typed existential types subtlety

Let τ_0 be $\exists\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha)$ and v_0 a value of type *bool*. Let v_1 and v_2 be two values of type τ_0 with incompatible witness types, *e.g.* $\lambda f.\,\lambda x.\,1 + (f\ (1 + x))$ and $\lambda f.\,\lambda x.\,not\ (f\ (not\ x))$.

Let v be the function $\lambda b.\,\text{if } b \text{ then } v_1 \text{ else } v_2$ of type $\textit{bool} \to \tau_0$.

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

We have $\varnothing \vdash a_1 : \exists\alpha.\,\alpha \to \alpha$ while $\varnothing \nvdash a_2 : \tau$.

What happened? 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.

Implicitly-typed existential types subtlety

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

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

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

Comments?

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

Implicitly-typed existential types subtlety

Existential types are trickier than they may appear at first.

The subject reduction property breaks if reduction is not restricted to expressions in head-normal forms.

Unrestricted reduction is still safe because well-typedness may eventually be recovered by further reduction steps—so that progress will never breaks.

Implicitly-typed existential types encoding

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

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

In the call-by-value setting, $\lambda k.\, [\![a]\!]\ k$ would come from the reduction of $[\![\textit{pack}\, a]\!]$, *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 that a_1 be reduced to some head normal form $\lambda k.\, [\![a]\!]\ k$.

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Iso-existential types in ML

What if one wished to extend ML with existential types?

Full type inference for existential types is undecidable, just like type inference for universals.

However, introducing existential types in ML is easy if one is willing to rely on user-supplied *annotations* that indicate *where* and *how* to pack and unpack.

Iso-existential types in ML

This *iso-existential* approach was suggested by Läufer and Odersky [1994].

Iso-existential types are explicitly *declared:*

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

This introduces two constants, with the following type schemes:

$$\begin{array}{rcl} \mathit{pack}_D & : & \forall\bar{\alpha}\bar{\beta}.\,\tau \to D\,\vec{\alpha} \\ \mathit{unpack}_D & : & \forall\bar{\alpha}\gamma.\,D\,\vec{\alpha} \to (\forall\bar{\beta}.\,(\tau \to \gamma)) \to \gamma \end{array}$$

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

Iso-existential types in ML

One point has been hidden on the previous slide. The “type scheme:”

$$\forall \bar{\alpha}\gamma.\, D\,\vec{\alpha} \to (\forall \bar{\beta}.\,(\tau \to \gamma)) \to \gamma$$

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

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

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

We have seen a version of this rule in System F earlier; this is an ML(-like) version.

The term M_2 must be polymorphic, which Gen can prove.

Iso-existential types in ML (type inference, skip)

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

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

The construct unpack_D leads to this constraint generation rule (see type inference):

$$\langle\!\langle \mathit{unpack}_D\ M_1\ M_2 : \tau_2 \rangle\!\rangle \quad = \quad \exists\bar{\alpha}.\left(\begin{array}{l} \langle\!\langle M_1 : D\ \vec{\alpha} \rangle\!\rangle \\ \forall\bar{\beta}.\ \langle\!\langle M_2 : \tau \to \tau_2 \rangle\!\rangle \end{array} \right)$$

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

A universally quantified constraint appears where polymorphism is *required.*

Iso-existential types in ML

In practice, Läufer and Odersky suggest fusing iso-existential types with algebraic data types.

This can be done in OCaml using GADTs (see last part of the course). The syntax for this in OCaml is:

$$\textit{type}\ D\ \vec{\alpha} = \ell : \tau\ \to D\ \vec{\alpha}$$

where ℓ is a data constructor and $\bar{\beta}$ appears free in τ but does not appear in $\vec{\alpha}$. The elimination construct is typed as:

$$\langle\!\langle \textit{match}\ M_1\ \textit{with}\ \ell\ x \to M_2 : \tau_2 \rangle\!\rangle \quad = \quad \exists \bar{\alpha}. \begin{pmatrix} \langle\!\langle M_1 : D\ \vec{\alpha} \rangle\!\rangle \\ \forall \bar{\beta}. \textit{def}\ x : \tau\ \textit{in}\ \langle\!\langle M_2 : \tau_2 \rangle\!\rangle \end{pmatrix}$$

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

An example

Define *Any* $\approx \exists\beta.\beta$. An attempt to extract the raw content of a package fails:

$$\begin{array}{rcl} \langle\!\langle \mathit{unpack}_{\mathit{Any}}\; M_1\; (\lambda x.\, x) : \tau_2 \rangle\!\rangle & = & \langle\!\langle M_1 : \mathit{Any} \rangle\!\rangle \wedge \forall\beta.\, \langle\!\langle \lambda x.\, x : \beta \rightarrow \tau_2 \rangle\!\rangle \\ & \Vdash & \forall\beta.\, \beta = \tau_2 \\ & \equiv & \mathit{false} \end{array}$$

(Recall that $\beta \# \tau_2$.)

An example

Define

$$D\ \alpha \approx \exists\beta.(\beta \to \alpha) \times \beta$$

A client that regards β as abstract succeeds:

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

Existential types calls for universal types!

Exercise Let *thunk* $\alpha \approx \exists\beta.(\beta \to \alpha) \times \beta$ be the type of frozen computations. Assume given a list l with elements of type *thunk* τ_1.

Assume given a function g of type $\tau_1 \to \tau_2$. Transform the list l into a new list l' of frozen computations of type *thunk* τ_2 (without actually running any computation).

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

Try generalizing this example to a function that receives g and l and returns l' : it does not typecheck...

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

In expression *let* α, x = *unpack* M_1 *in* M_2, occurrences of x in M_2 can only be passed to external functions (free variables) that are polymorphic in α so that α does not leak out of its context.

Limits of iso-encodings

Using datatypes for existential and especially universal types is a simple solution to make them compatible with ML, but it comes with some limitations:

- All types must be declared before being used
- Programs become quite verbose, with many constructors that amount to writting type annotations, but in a more rigid way
- In particular, there is no canonical way of representing them. For exemple, a thunk of type $\exists\beta(\beta \to \mathit{int}) \times \beta$ could have been defined as Delay (succ, 1) where Delay is either one of

```
type int_thunk = Delay : ('b → int) ∗ 'b → int_thunk
type 'a thunk = Delay : ('b → 'a) ∗ 'b → 'a thunk
```

  but the two types are incompatible.

Hence, other primitive solutions have been considered, especially for universal types.

Uses of existential types

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

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

specifies an abstract implementation of integer stacks.

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

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

Rossberg, Russo, and Dreyer show that after all, generative modules can be encoded into System F with existential types [Rossberg et al., 2014].

Existential types in OCaml

Existential types are available indirectly in OCaml as a degenerate case of GADT and via abstract types and first-class modules.

Via GADT (iso-existential types)

```
type 'a thunk = Delay : ('b → 'a) * 'b → 'a thunk
let freeze f x = Delay (f, x)
let unfreeze (Delay (f, x)) = f x
```

Via first-class modules (abstract types)

```
module type Thunk = 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
   : Delay)
let unfreeze (type u) (module M : Thunk with type a = u) = M.f M.x
```

Contents

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

An introduction to GADTs

What are they?

ADTs

Types of constructors are surjective: all types can potentially be reached

type α list =
  | Nil : α list
  | Const : $\alpha * \alpha$ list $\rightarrow \alpha$ list

GADTs

This is no more the case with GADTs

type (α, β) eq =
  | *Eq* : (α, α) eq
  | Any : (α, β) eq

The *Eq* constructor may only build values of types of (α, α) eq.
For example, it cannot build values of type (int, string) eq.

The criteria is *per constructor*: it remains a GADT when another (even *regular*) constructor is added.

Examples   Defunctionalization

```
let add (x, y) = x + y in
let not x = if x then false else true in
let body b =
    let step x =
      add (x, if not b then 1 else 2)
    in step (step 0))
in body true
```

Introduce a constructor per function

```
type (_, _) apply =
  | Fadd   : (int ∗ int, int) apply
  | Fnot   : (bool, bool) apply
  | Fbody : (bool, int) apply
  | Fstep  : bool → (int, int) apply
```

Define a single apply function that dispatches all function calls:

```
let rec apply : type a b. (a, b) apply → a → b = fun f arg →
  match f with
  | Fadd      → let x, y = arg in x + y
  | Fnot      → let x = arg in if x then false else true
  | Fstep b   → let x = arg in
                apply Fadd (x, if apply Fnot b then 1 else 2)
  | Fbody     → let b = arg in
                apply (Fstep b) (apply (Fstep b) 0)
in apply Fbody true
```

Examples Typed evaluator

A typed abstract-syntax tree

```
type _ expr =
  | Int     : int → int expr
  | Zerop   : int expr → bool expr
  | If      : (bool expr ∗ α expr ∗ α expr) → α expr
let e0 : int expr = (If (Zerop (Int 0), Int 1, Int 2))
```

A typed evaluator (with no failure)

```
let rec eval : type a . a expr → a = fun x → match x with
  | Int x          → x                                    (∗ a = int  ∗)
  | Zerop x        → eval x > 0                           (∗ a = bool ∗)
  | If (b, e1, e2) → if eval b then eval e1 else eval e2
let b0 = eval e0
```

Exercise

What would you have to do without GADTs? Define a typed abstract syntax tree for the simply-typed λ-calculus and a *typed* evaluator.

Examples

Generic programming

Example of printing

```
type _ ty =
  | Tint : int ty
  | Tbool : bool ty
  | Tlist : α ty → (α list) ty
  | Tpair : α ty ∗ β ty → (α ∗ β) ty

let rec to_string : type a. a ty → a → string = fun t x → match t with
  | Tint → string_of_int x
  | Tbool → if x then "true" else "false"
  | Tlist t → "[" ^ String.concat "; " (List.map (to_string t) x) ^ "]"
  | Tpair (a, b) →
      let u, v = x in "(" ^ to_string a u ^ ", " ^ to_string b v ^ ")"

let s = to_string (Tpair (Tlist Tint, Tbool)) ([1; 2; 3], true)
```

Examples

Encoding sum types

```
type (α, β) sum = Left of α | Right of β
```

can be encoded as a product:

```
type (_, _, _) tag  = Ltag : (α, α, β) tag | Rtag : (β, α, β) tag
type (α, β) prod  = Prod : (γ, α, β) tag * γ → (α, β) prod

let sum_of_prod (type a b) (p : (a, b) prod) : (a, b) sum =
  let Prod (t, v) = p in match t with Ltag → Left v | Rtag → Right v
```

Prod is a single, hence superfluous constructor: it need not be allocated.

A field common to both cases can be accessed without looking at the tag!

```
type (α, β) prod = Prod : (γ, α, β) tag * γ * bool → (α, β) prod
let get (type a b) (p : (a, b) prod) : bool =
  let Prod (t, v, s) = p in s
```

Examples

Encoding sum types

Exercise

Specialize the encoding of sum types to the encoding of 'a list

Other uses of GADTs

GADTs

- May encode data-structure invariants, such as the state of an automaton, as illustrated by Pottier and Régis-Gianas [2006] for typechecking LR-parsers.
- They may be used to implement a form of dynamic type (similarly to the generic printer)
- They may be used to optimize representation (*e.g.* sum's encoding)
- GADTs can be used to encode type classes, using a technique analogous to defunctionalization [Pottier and Gauthier, 2006].

Reducing GADTs to type equality (and existential types)

All GADTs can be encoded with a single one, encoding type equality:

```
type (α, β) eq = Eq : (α, α) eq
```

For instance, generic programming can then be redefined as follows:

```
type α ty =
  | Tint  : (α, int) eq → α ty                                  (* int ty          *)
  | Tlist : (α, β list) eq * β ty → α ty                        (* α ty → α list ty *)
  | Tpair : (α, (β * γ)) eq * β ty * γ ty → α ty
```

This declaration is not a GADT, just an existential type!

- ▷ We enlarge the domain of each constructor,
- ▷ But require a proof evidence as an extra argument that a certain equality holds *to restrict the possible uses* of the constructors.

```
let rec to_string : type a. a ty → a → string = fun t x → match t with
  | Tint Eq        → string_of_int x
  | Tlist (Eq, l)  → "[" ^String.concat "; " (List.map (to_string l) x)^ "]"
  | Tpair (Eq,a,b) →
      let u, v = x in "(" ^ to_string a u ^ ", " ^ to_string b v ^ ")"

let s = to_string (Tpair (Eq, Tlist (Eq, Tint Eq), Tint Eq)) ([1; 2; 3], 0)
```

Reducing GADTs to type equality (and existential types)

All GADTs can be encoded with a single one :

```
type (α, β) eq = Eq : (α, α) eq
```

For instance, generic programming can be redefined as follows:

```
type α ty =
  | Tint  : (α, int) eq → α ty
  | Tlist : (α, β list) eq ∗ β ty → α ty
  | Tpair : (α, (β ∗ γ)) eq ∗ β ty ∗ γ ty → α ty
```

This declaration is not a GADT, just an existential type!

```
let rec to_string : type a. a ty → a → string = fun t x → match t with
 | Tint Eq → string_of_int x
 | Tlist (Eq, l) → ...
 | Tpair (Eq, a, b) → ...
```

▷ Pattern "Tint *Eq*" is GADT matching

Reducing GADTs to type equality (and existential types)

All GADTs can be encoded with a single one :

```
type (α, β) eq = Eq : (α, α) eq
```

For instance, generic programming can be redefined as follows:

```
type α ty =
  | Tint  : (α, int) eq → α ty
  | Tlist : (α, β list) eq * β ty → α ty
  | Tpair : (α, (β * γ)) eq * β ty * γ ty → α ty
```

This declaration is not a GADT, just an existential type!

```
let rec to_string : type a. a ty → a → string = fun t x → match t with
 | Tint p → let Eq = p in string_of_int x
 | Tlist (Eq, l) → ...
 | Tpair (Eq, a, b) → ...
```

- ▷ Pattern “Tint *Eq*” is GADT matching
- ▷ **let** *Eq* = p **in**.. introduces the equality a = int in the current branch

Formalisation of GADTs

We can extend System F with type equalities to encode GADTs.

We *cannot* encode type equalities in System F—but in System F^ω: they bring something more, namely *local equalities* in the typing context.

We write $\tau_1 \sim \tau_2$ for (τ_1, τ_2) eq

When typechecking an expression

$$E[\textit{let } x : \tau_1 \sim \tau_2 = M_0 \textit{ in } M] \qquad E[\lambda x : \tau_1 \sim \tau_2 .\, M]$$

- M is typechecked with the asumption that $\tau_1 \sim \tau_2$, *i.e.* types τ_1 and τ_2 are equivalent, which allows for type conversion within M
- but E and M_0 are typechecked without this asumption
- What is learned by an equation remains local to its static scope, and does not extend to its surrounding context (or the rest of the program execution trace).

Fc (simplified) *Add equality coercions to System F*

Types

$$\tau ::= \ldots \mid \tau_1 \sim \tau_2$$

Expressions

$$M ::= \ldots \mid \gamma \triangleleft M \mid \gamma$$

Coercions are first-class and can be applied to terms.

Coercions witness type equivalences:

$\gamma ::= \alpha$	variable
$\mid \langle\tau\rangle$	reflexivity
$\mid \mathsf{sym}\,\gamma$	symmetry
$\mid \gamma_1 ; \gamma_2$	transitivity
$\mid \gamma_1 \to \gamma_2$	arrow coercions
$\mid \mathsf{left}\,\gamma$	left projection
$\mid \mathsf{right}\,\gamma$	right projection
$\mid \forall\alpha.\gamma$	type generalization
$\mid \gamma @ \tau$	type instantiation

Typing rules:

Coerce
$$\frac{\Gamma \vdash M : \tau_1 \qquad \Gamma \vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash \gamma \triangleleft M : \tau_2}$$

Coercion
$$\frac{\Gamma \Vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash \gamma : \tau_1 \sim \tau_2}$$

Coabs
$$\frac{\Gamma, x : \tau_1 \sim \tau_2 \vdash M : \tau}{\Gamma \vdash \lambda x : \tau_1 \sim \tau_2 . M : \tau_1 \sim \tau_2 \to \tau}$$

Fc (simplified) Typing of coercions

Eq-Hyp
$$\frac{y : \tau_1 \sim \tau_2 \in \Gamma}{\Gamma \Vdash y : \tau_1 \sim \tau_2}$$

Eq-Ref
$$\frac{\Gamma \vdash \tau}{\Gamma \Vdash \langle\tau\rangle : \tau \sim \tau}$$

Eq-Sym
$$\frac{\Gamma \Vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \Vdash \mathsf{sym}\,\gamma : \tau_2 \sim \tau_1}$$

Eq-Trans
$$\frac{\Gamma \Vdash \gamma_1 : \tau_1 \sim \tau \qquad \Gamma \Vdash \gamma_2 : \tau \sim \tau_2}{\Gamma \Vdash \gamma_1 ; \gamma_2 : \tau_1 \sim \tau_2}$$

Eq-Arrow
$$\frac{\Gamma \Vdash \gamma_1 : \tau_1' \sim \tau_1 \qquad \Gamma \Vdash \gamma_2 : \tau_2 \sim \tau_2'}{\Gamma \Vdash \gamma_1 \to \gamma_2 : \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

Eq-Left
$$\frac{\Gamma \Vdash \gamma : \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}{\Gamma \Vdash \mathsf{left}\,\gamma : \tau_1' \sim \tau_1}$$

Eq-Right
$$\frac{\Gamma \Vdash \gamma : \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}{\Gamma \Vdash \mathsf{right}\,\gamma : \tau_2 \sim \tau_2'}$$

Eq-All
$$\frac{\Gamma, \alpha \Vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \Vdash \forall\alpha.\,\gamma : \forall\alpha.\,\tau_1 \sim \forall\alpha.\,\tau_2}$$

Eq-Inst
$$\frac{\Gamma \Vdash \gamma : \forall\alpha.\,\tau_1 \sim \forall\alpha.\,\tau_2 \qquad \Gamma \vdash \tau}{\Gamma \Vdash \gamma @ \tau : [\alpha \mapsto \tau]\tau_1 \sim [\alpha \mapsto \tau]\tau_2}$$

Only equalities between *injective* type constructors can be decomposed.

Semantics

Coercions should be without computational content

- they are just type information, and should be erased at runtime
- they should not block redexes
- in Fc, we may always push them down inside terms, adding new reduction rules:

$$\begin{array}{rcl}
(\gamma \triangleleft V_1)\, V_2 & \longrightarrow & \mathsf{right}\,\gamma \triangleleft (V_1\ (\mathsf{left}\,\gamma \triangleleft V_2)) \\
(\gamma \triangleleft V)\ \tau & \longrightarrow & (\gamma @ \tau) \triangleleft (V\ \tau) \\
\gamma_1 \triangleleft (\gamma_2 \triangleleft V) & \longrightarrow & (\gamma_1 ; \gamma_2) \triangleleft V
\end{array}$$

Semantics

Coercions should be without computational content

Except for coercion abstractions that must stop the evaluation

- Otherwise, one could attempt to reduce M in $\lambda \mathit{int} \sim \mathit{bool}.\, M$ when M is *not* $(\mathit{bool} \lhd 0)$, which is well-typed in this context.
- In call-by-value,

$\lambda x : \tau_1 \sim \tau_2.\, M$	freezes	the evaluation of M,
$M \lhd \gamma$	resumes	the evaluation of M.

  Must always be enforced, even with other strategies
- Full reduction *at compile time* may still be perfomed, but be aware of stuck programs and treat them as dead branches.

Type soundness *Syntactic proofs*

Type soundness

By subject reduction and progress with explicit coercions

Erasing semantics

Important and not so obvious.

$\gamma \triangleleft M$	erases	to M
γ	erases	to $\diamond$

Slogan that “coercion have 0-bit information”, *i.e.*
Coercions need not be passed at runtime—-but still block the reduction.
Expressions and typing rules.

$$\textsc{Coerce}\quad \frac{\Gamma \vdash M : \tau_1 \qquad \Gamma \vdash \diamond : \tau_1 \sim \tau_2}{\Gamma \vdash M : \tau_2}$$

$$\textsc{Coercion}\quad \frac{\Gamma \Vdash \tau_1 \sim \tau_2}{\Gamma \vdash \diamond : \tau_1 \sim \tau_2}$$

$$\textsc{CoAbs}\quad \frac{\Gamma, x : \tau_1 \sim \tau_2 \vdash M : \tau}{\Gamma \vdash \lambda x : \tau_1 \sim \tau_2 .\, M : \tau_1 \sim \tau_2 \to \tau}$$

Type soundness *Syntactic proofs*

The introduction of type equality constraints in System F has been introduced and formalized by Sulzmann et al. [2007].

Scherer and Rémy [2015] show how strong reduction and confluence can be recovered in the presence of possibly uninhabited coercions.

Type soundness *Semantic proofs*

Equality coercions are a small logic of type conversions.

Type conversions may be enriched with more operations.

A very general form of coercions has been introduced by Cretin and Rémy [2014].

The type soundness proof became too cumbersome to be conducted syntactically.

Instead a semantic proof is used, interpreting types as sets of terms (a technique similar to unary logical relations)

Type checking / inference

With explicit coercions, types are fully determined from expressions.

However, the user prefers to leave applications of COERCE implicit.

Then types becomes ambiguous: when leaving the scope of an equation: which form should be used, among the equivalent ones?

This must be determined from the context, including the return type, and calls for extra type annotations:

```
let rec eval : type a . a expr → a = fun x → match x with
 | Int x          → x   (* x : int, but a = int, should we return x : a? *)
 | Zerop x        → eval x > 0
 | If (b, e1, e2) → if eval b then eval e1 else eval e2
```

In ML, type annotations must be used to tell

- the type of the context
- which datatypes must be typed as GADTs.

In Coq, one must use return type annotations on matches.

Type inference in ML-like languages with GADTs

Simonet and Pottier [2007] gave a presentation of type inference for GADTs with general typing constraints for ML-like languages.

Pottier and Régis-Gianas [2006] introduced a stratified approach to better propagate constraints from outisde to inside GADTs contexts.

Vytiniotis et al. [2011] introduced the outside-in approach, used in Haskell, which restricts type information to flow from outside to inside GADT contexts.

Garrigue and Rémy [2013] introduced the notion of ambivalent types, used in OCaml, to restrict type occurrences that must be considered ambiguous and explicitly specified using type annotations.

Contents

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Type-preserving compilation

Compilation is type-preserving when each intermediate language is *explicitly typed*, and each compilation phase transforms a typed program into a typed program in the next intermediate language.

Why *preserve types* during compilation?

- it can help debug the compiler;
- types can be used to drive optimizations;
- types can be used to produce *proof-carrying code;*
- proving that types are preserved can be the first step towards proving that the *semantics* is preserved [Chlipala, 2007].

Type-preserving compilation

Type-preserving compilation exhibits an encoding of programming constructs into programming languages with usually richer type systems.

The encoding may sometimes be used directly as a programming idiom in the source language.

For example:

- Closure conversion requires an extension of the language with existential types, which happens to be very useful on their own.
- Closures are themselves a simple form of objects, which can also be explained with existential types.
- Defunctionalization may be done manually on some particular programs, *e.g.* in web applications to monitor the computation.

Type-preserving compilation

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

- *CPS conversion* fixes the order of evaluation, names intermediate computations, and makes all function calls tail calls;
- *closure conversion* makes environments and closures explicit, and produces a program where all functions are closed;
- allocation and initialization of tuples is made explicit;
- the calling convention is made explicit, and variables are replaced with (an unbounded number of) machine registers.

Translating types

In general, a type-preserving compilation phase involves not only a translation of *terms,* mapping 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.

See the old lecture on type closure conversion.

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 moved and applied in another runtime environment.

Closures can also be used to represent recursive functions and objects (in the object-as-record-of-methods paradigm).

Source and target

In the following,

- the *source* calculus has *unary* λ-abstractions, which can have free variables;
- the *target* calculus has *binary* λ-abstractions, which must be *closed.*

Closure conversion can be easily extended to n-ary functions, or n-ary functions may be *uncurried* in a separate, type-preserving compilation pass.

Variants of closure conversion

There are at least two variants of closure conversion:

- in the *closure-passing variant,*
  the closure and the environment are a single memory block;
- in the *environment-passing variant,*
  the environment is a separate block, to which the closure points.

The impact of this choice on the translation of terms is minor.

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

Closure-passing closure conversion

Let $\{x_1, \dots, x_n\}$ be $\mathrm{fv}(\lambda x.\, a)$:

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

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

Important! The layout of the environment must be known only at the closure allocation site, not at the call site. In particular, *proj*$_0$ clo need not know the size of clo.

Environment-passing closure conversion

Let $\{x_1, \ldots, x_n\}$ be $\mathrm{fv}(\lambda x.\, a)$:

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

Questions: 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, $[\![\tau_1 \to \tau_2]\!]$?

Environment-passing closure conversion

Let $\{x_1, \ldots, x_n\}$ be $\mathrm{fv}(\lambda x.\, a)$:

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

Assume $\Gamma \vdash \lambda x.\, a : \tau_1 \to \tau_2$.
Assume, *w.l.o.g.*. $\mathrm{dom}(\Gamma) = \mathrm{fv}(\lambda x.\, a) = \{x_1, \ldots, x_n\}$.

Write $[\![\Gamma]\!]$ for the tuple type $x_1 : [\![\tau'_1]\!]; \ldots ; x_n : [\![\tau'_n]\!]$ where Γ is $x_1 : \tau'_1; \ldots ; x_n : \tau'_n$. We also use $[\![\Gamma]\!]$ as a type to mean $[\![\tau'_1]\!] \times \ldots \times [\![\tau'_n]\!]$.

We have $\Gamma, x : \tau_1 \vdash a : \tau_2$, so in environment $[\![\Gamma]\!], x : [\![\tau_1]\!]$, we have

- env has type $[\![\Gamma]\!]$,
- $code$ has type $([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!]$, and
- the entire closure has type $(([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times [\![\Gamma]\!]$.

Now, *what should be the definition of $[\![\tau_1 \to \tau_2]\!]$?*

Towards a type translation

Can we adopt this as a definition?

$$[\![\tau_1 \rightarrow \tau_2]\!] \quad = \quad (([\![\Gamma]\!] \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!]) \times [\![\Gamma]\!]$$

Naturally not. This definition is mathematically ill-formed: we cannot use Γ out of the blue.

That is, this definition is not uniform: it depends on Γ, *i.e.* the size and layout of the environment.

Do we really need to have a uniform translation of types?

Towards a type translation

Yes, we do.

We need a uniform translation of types, not just because it is nice to have one, but because it describes a *uniform calling convention.*

If closures with distinct environment sizes or layouts receive distinct types, then we will be unable to translate this well-typed code:

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

Furthermore, we want function invocations to be translated uniformly, without knowledge of the size and layout of the closure's environment.

So, *what could be the definition of* $[\![\tau_1 \rightarrow \tau_2]\!]$*?*

The type translation

The only sensible solution is:

$$[\![\tau_1 \to \tau_2]\!] \quad = \quad \exists\alpha.((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha$$

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 type translation

The existential quantification also provides a form of *security*: the caller cannot do anything with the environment except pass it as an argument to the code; in particular, it cannot inspect or modify the environment.

For instance, in the source language, the following coding style guarantees that x remains even, no matter how f is used:

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

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].

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Typed closure conversion

Everything is now set up to prove that, in System F with existential types:

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

Environment-passing closure conversion

Assume $\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2$ and $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \mathrm{fv}(\lambda x.\, M)$.

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

We find $[\![\Gamma]\!] \vdash [\![\lambda x{:}\tau_1.\, M]\!] : [\![\tau_1 \to \tau_2]\!]$, as desired.

Environment-passing closure conversion

Assume $\Gamma \vdash M : \tau_1 \to \tau_2$ and $\Gamma \vdash M_1 : \tau_1$.

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

We find $[\![\Gamma]\!] \vdash [\![M\ M_1]\!] : [\![\tau_2]\!]$, as desired.

Environment-passing closure conversion recursion

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

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

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

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

What is the weak point of this variant?

A new closure is allocated at every call.

Environment-passing closure conversion recursion

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

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

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

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

Environment-passing closure conversion recursion

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

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

Problem?

Environment-passing closure conversion recursion

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

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.

Environment-passing closure conversion recursion

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

The encoding is simple.

However, this requires the introduction of recursive non-functional values "let rec x = v". While this is a useful construct, it really alters the operational semantics and requires updating the type soundness proof.

- Algebraic Data Types
  - Equi- and iso- recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes
- Application to typed closure conversion
  - Environment passing
  - Closure passing

Closure-passing closure conversion

$$
\begin{array}{rcl}
[\![\lambda x.\, M]\!] & = & \textit{let } code = \lambda(clo, x). \\
& & \quad \textit{let } (\_, x_1, \ldots, x_n) = clo \textit{ in} \\
& & \quad [\![M]\!] \\
& & \textit{in } (code, x_1, \ldots, x_n) \\
& & \\
[\![M_1\ M_2]\!] & = & \textit{let } clo = [\![M_1]\!] \textit{ in} \\
& & \textit{let } code = \textit{proj}_0\ clo \textit{ in} \\
& & code\ (clo, [\![M_2]\!])
\end{array}
$$

There are two difficulties:

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

Closure-passing closure conversion

There are two difficulties:

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

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

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

Tuples, rows, row variables

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

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

The notation $(\tau_1 \times \ldots \times \tau_n)$ was sugar for $\Pi\,(\tau_1; \ldots; \tau_n; \epsilon)$.

Let us now 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.

Typing rules for tuples

The typing rules for tuple construction and deconstruction are:

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

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:*

$$\textit{proj}_i : \forall \alpha._1 \ldots \alpha_i \rho.\ \Pi\,(\alpha_1; \ldots; \alpha_i; \rho) \to \alpha_i$$

About Rows

Rows were invented by Wand and improved by RÃ©my in order to ascribe precise types to operations on *records.*

The case of tuples, presented here, is simpler.

Rows are used to describe *objects* in Objective Caml [Rémy and Vouillon, 1998].

Rows are explained in depth by Pottier and RÃ©my [Pottier and Rémy, 2005].

Closure-passing closure conversion

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

$$
\begin{array}{rll}
 & [\![\tau_1 \rightarrow \tau_2]\!] & \\
= & \exists \rho. & \rho \text{ describes the environment} \\
 & \quad \mu\alpha. & \alpha \text{ is the concrete type of the closure} \\
 & \quad\quad \Pi\,(& \text{a tuple...} \\
 & \quad\quad\quad (\alpha \times [\![\tau_1]\!]) \rightarrow [\![\tau_2]\!]; & \text{...that begins with a code pointer...} \\
 & \quad\quad\quad \rho & \text{...and continues with the environment} \\
 & \quad\quad) &
\end{array}
$$

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

Question: Why is it $\exists\rho.\ \mu\alpha.\ \tau$ 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.

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

Closure-passing closure conversion

Let $Clo(R)$ abbreviate $\mu\alpha.\Pi\,((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R)$.

Let $UClo(R)$ abbreviate its unfolded version,
$\Pi\,((Clo(R) \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R)$.

We have $[\![\tau_1 \to \tau_2]\!] = \exists\rho.Clo(\rho)$.

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

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

Closure-passing closure conversion recursive functions

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

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

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

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:

Check well-typedness with monomorphic recursion.

Closure-passing closure conversion recursive functions

The problem to adapt this encoding to polymorphic recursion is that recursive occurrences of f are rebuilt from the current invocation of the closure, *i.e.* is monomorphic 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 invokation, *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.

Closure-passing closure conversion recursive functions

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

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

Remind that $Clo(R)$ abbreviates $\mu\alpha.\Pi\ ((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R)$. Hence, $\vec{\beta}$ are free variables of $Clo(R)$.

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.

Mutually recursive functions Environment passing

Can we compile mutually recursive functions?

$$M \triangleq \mu(f_1, f_2).(\lambda x_1.\, M_1, \lambda x_2.\, M_2)$$

Environment passing:

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

Mutually recursive functions — Environment passing

Can we compile mutually recursive functions?

$$M \triangleq \mu(f_1, f_2).(\lambda x_1.\, M_1, \lambda x_2.\, M_2)$$

Environment passing:

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

Mutually recursive functions Environment passing

Can we compile mutually recursive functions?

$$M \triangleq \mu(f_1, f_2).(\lambda x_1.\, M_1, \lambda x_2.\, M_2)$$

Environment passing:

```
let $code_i$ = $\lambda(clo, x)$.
    let $(\_\,, f_1, f_2, x_1, \ldots, x_n)$ = $clo$ in $[\![M_i]\!]$
 in
let rec $clo_1$ = $(code_1, clo_1, clo_2, x_1, \ldots, x_n)$
    and $clo_2$ = $(code_2, clo_1, clo_2, x_1, \ldots, x_n)$
 in $clo_1, clo_2$
```

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

Mutually recursive functions — Environment passing

Can we compile mutually recursive functions?

$$M \triangleq \mu(f_1, f_2).(\lambda x_1.\, M_1, \lambda x_2.\, M_2)$$

Environment passing:

```
let $code_1 = \lambda(clo, x)$.
    let $(\_code_1, \_code_2, f_1, f_2, x_1, \ldots, x_n) = clo$ in $[\![M_1]\!]$ in
let $code_2 = \lambda(clo, x)$.
    let $(\_code_2, f_1, f_2, x_1, \ldots, x_n) = clo$ in $[\![M_2]\!]$ in
let rec $clo_1 = (code_1, code_2, clo_1, clo_2, x_1, \ldots, x_n)$ and $clo_2 = clo_1.tail$
 in $clo_1, clo_2$
```

- $clo_1.tail$ returns a pointer to the tail $(code_2, clo_1, clo_2, x_1, \ldots, x_n)$ of clo_1 without allocating a new tuple.
- This is only possible with some support from the GC (and extra-complexity and runtime cost for GC)

Optimizing representations

Can closure passing and environment passing be mixed?

No because the calling-convention (*i.e.*, the encoding of application) must be uniform.

However, their is some flexibility in the representation of the closure. For instance, the following change is completely local:

$$\begin{array}{rcl} [\![\lambda x.\, M]\!] & = & \textit{let code} = \lambda(clo, x). \\ & & \quad \textit{let}\ (\_, (x_1, \ldots, x_n)) = clo\ \textit{in}\ [\![M]\!]\ \textit{in} \\ & & (code, (x_1, \ldots, x_n)) \\ [\![M_1\ M_2]\!] & = & \textit{let}\ clo = [\![M_1]\!]\ \textit{in} \\ & & \textit{let code} = \textit{proj}_0\ clo\ \textit{in} \\ & & code\ (clo, [\![M_2]\!]) \end{array}$$

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

Encoding of objects

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

A class definition is an object generator:

$$\begin{array}{l} class\, c\,(x_1, \dots x_q)\{ \\ \qquad \textit{meth}\;\; m_1 = M_1 \\ \qquad \dots \\ \qquad \textit{meth}\;\; m_p = M_p \\ \} \end{array}$$

Given arguments for parameter x_1, $\dots x_1$, it will build recursive methods m_1, $\dots m_n$.

Encoding of objects

A class can be compiled into an object closure:

$$\begin{array}{l} \textit{let}\ m = \\ \qquad \textit{let}\ m_1 = \lambda(m, x_1, \ldots, x_q).\, M_1\ \textit{in} \\ \qquad \ldots \\ \qquad \textit{let}\ m_p = \lambda(m, x_1, \ldots, x_q).\, M_p\ \textit{in} \\ \qquad \{m_1, \ldots, m_p\}\ \ \textit{in} \\ \lambda x_1 \ldots x_q.\, (m, x_1, \ldots x_q) \end{array}$$

Each m_i is bound to the code for the corresponding method.
The code of all methods are combined into a record of methods, which is shared between all objects of the same class.

Calling method m_i of an object p is

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

How can we type the encoding?

Typed encoding of objects

Let τ_i be the type of M_i, and row R describe the types of $(x_1, \dots x_q)$.

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

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

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

Calling a method of an object p of type M is

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

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

Typed encoding of objects

Typed encoding of objects were first studied in the 90's to understand what objects really are in a type setting.

These encodings are in fact type-preserving compilation of (primitive) objects.

There are several variations on these encodings. See [Bruce et al., 1999] for a comparison.

See [Rémy, 1994] for an encoding of objects in (a small extension of) ML with iso-existentials and universals.

See [Abadi and Cardelli, 1996, 1995] for more details on primitive objects.

Moral of the story

Type-preserving compilation is rather *fun.* (Yes, really!)

It forces compiler writers to make the structure of the compiled program *fully explicit,* in type-theoretic terms.

In practice, building explicit type derivations, ensuring that they remain small and can be efficiently typechecked, can be a lot of work.

Optimizations

Because we have focused on type preservation, we have studied only naÃ¯ve closure conversion algorithms.

More ambitious versions of closure conversion require program analysis: see, for instance, Steckler and Wand [1997]. These versions *can* be made type-preserving.

Other challenges

Defunctionalization, an alternative to closure conversion, offers an interesting challenge, with a simple solution [Pottier and Gauthier, 2006].

Designing an efficient, type-preserving compiler for an *object-oriented language* is quite challenging. See, for instance, Chen and Tarditi [2005].

Bibliography I

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

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

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

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

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

Bibliography II

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

▷ 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.

Julien Cretin and Didier Rémy. System F with Coercion Constraints. In *Logics In Computer Science (LICS)*. ACM, July 2014.

Jacques Garrigue and Didier Rémy. Ambivalent Types for Principal Type Inference with GADTs. In *11th Asian Symposium on Programming Languages and Systems*, Melbourne, Australia, December 2013.

Bibliography III

▷ Nadji Gauthier and François Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 150–161, September 2004. doi: http://doi.acm.org/10.1145/1016850.1016872.

▷ Neal Glew. A theory of second-order trees. In Daniel Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2002. doi: $10.1007/3\text{-}\sigma_2 540\text{-}\sigma_2 45927\text{-}\sigma_2 8\backslash\_11$.

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.

Bibliography IV

▷ 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.

Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.

Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17, May 2007.

▷ 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.

Bibliography V

- 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.
- Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.
- François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.
- François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 232–244, January 2006.

Bibliography VI

▷ François Pottier and Yann Régis-Gianas. Towards efficient, typed LR parsers. In *ACM Workshop on ML*, volume 148-2 of *Electronic Notes in Theoretical Computer Science*, pages 155–180, March 2006.

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

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

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

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

▷ Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. *J. Funct. Program.*, 24(5):529–607, 2014. doi: 10.1017/S0956796814000264.

Bibliography VII

▷ Gabriel Scherer and Didier Rémy. Full reduction in the face of absurdity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 685–709, 2015. doi: $10.1007/978\text{-}\sigma_2 3\text{-}\sigma_2 662\text{-}\sigma_2 46669\text{-}\sigma_2 8\_28$.

▷ Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Program. Lang. Syst.*, 29(1), January 2007. ISSN 0164-0925. doi: 10.1145/1180475.1180476.

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

▷ Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM. ISBN 1-59593-393-X. doi: 10.1145/1190315.1190324.

Bibliography VIII

▷ Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, September 2011. ISSN 0956-7968. doi: 10.1017/S0956796811000098.