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

Didier Rémy

October 27, 2017

## Plan of the course

Metatheory of System F

ADTs, Existential types, GATDs

## Abstract Data types, Existential types, GADTs

## Contents

## 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 (implicit version for conciseness):

- 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 $d_G\,(C_i\;v)\;v_1\;\ldots v_n \;\leadsto\; v_i\;v$

### 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 $\tau_i$

Algebraic datatypes introduce *iso-recursive types*.

- Algebraic Data Types
  - Equi- and iso-recursive types
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Generalized Algebraic Datatypes

## 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 iso-recursive 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:

$$\mathit{list}\,\alpha \quad \diamond \quad \mathit{unit} + \alpha \times \mathit{list}\,\alpha$$

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

## Equi- versus iso-recursive types

There are two standard approaches to recursive types, dubbed the *equi-recursive* and *iso-recursive* approaches.

In the equi-recursive approach, a recursive type is *equal* to its unfolding.

In the iso-recursive approach, a recursive type and its unfolding are related via explicit *coercions*.

## Equi-recursive types

In the equi-recursive 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 equi-recursive types

If desired, it is possible to use *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 $\mu\alpha.\tau$, 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 $\tau$ is *contractive,* that is, $\tau$ is a type constructor application or a forall introduction.

For instance, the type of lists of elements of type $\alpha$ is:

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

## Finite syntax for equi-recursive types

**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 by unification. (The latter approach is simpler, faster, and extends to the type inference problem.)

**In the presence of quantifiers** The situation is more subtle because of $\alpha$-conversion. A 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].

## Finite syntax for equi-recursive types

One can also prove [Brandt and Henglein, 1998] that equality is the least congruence generated by the following two rules:

$$\textsc{Fold/Unfold} \quad \mu\alpha.\tau = [\alpha \mapsto \mu\alpha.\tau]\tau \qquad\qquad \textsc{Uniqueness} \quad \frac{\tau_1 = [\alpha \mapsto \tau_1]\tau \qquad \tau_2 = [\alpha \mapsto \tau_2]\tau}{\tau_1 = \tau_2}$$

In both rules, $\tau$ must be contractive.

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

There is also a simple co-inductive definition:

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

### 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?*

## Type soundness for equi-recursive types

In the presence of equi-recursive 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 $\vec{\tau}_1 = \vec{\tau}_2$—this was used in the proof of Subject Reduction.

It remains true that $\mathsf{F}_1\ \vec{\tau}_1 = \mathsf{F}_2\ \vec{\tau}_2$ implies $\mathsf{F}_1 = \mathsf{F}_2$—this was used the proof of Progress.

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

(Exercise: prove type soundness for the *simply-typed $\lambda$-calculus* in Coq. Then, change the syntax of types from `Inductive` to `CoInductive`.)

## Iso-recursive types

With iso-recursive types, the folding/unfolding is witnessed by an explicit coercion (*e.g.* as above).

*The uniqueness rule is usually not present (hence, the equality relation is weaker).*

### Encoding iso-recursive types with ADT

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

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

The constructor $C$ coerces $[\beta \mapsto G\,\vec{\alpha}]\tau$ to $G\,\vec{\alpha}$ and the reverse coercion is the function $\lambda x.\,d_G\,x\,(\lambda y.\,y)$.
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* $G_\Pi$,
- $1$ *constructor* $C_\Pi : \forall \vec{\alpha}.\, \tau_1 \to \dots \tau_n \to G\,\vec{\alpha}$
- $n$ *destructors* $d_{\ell_i} : \forall \vec{\alpha}.\, G\,\vec{\alpha} \to \tau_i$
- $n$ *reduction rules* $d_{\ell_i}(C_\Pi\ v_1\ \dots v_n) \leadsto v_i$

*Can a record also be used for defining recursive types?*
*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]

## Regular ADTs

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

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

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

They usually need polymorphic recursion to be manipulated.

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

## Contents

## 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:

$$\begin{array}{ll}
= \quad \exists \rho . & \rho \text{ describes the state} \\
\qquad \mu \alpha . & \alpha \text{ is the concrete type of the closure} \\
\qquad\quad \Pi\, ( & \text{a tuple...} \\
\qquad\qquad \{ (\alpha \times \tau_1) \to \tau'_1 ; & \text{... that begins with a record...} \\
\qquad\qquad\quad \ldots & \\
\qquad\qquad\quad (\alpha \times \tau_n) \to \tau'_n \ \} \ ; & \text{... of method code pointers...} \\
\qquad\qquad \rho & \text{...and continues with the state} \\
\qquad\quad ) & \quad \text{(a tuple of unknown length)}
\end{array}$$

## 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'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:

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

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

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

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

Note the *imperfect* duality between universals and existentials:

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

## 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 $\tau$ for some *unknown* $\alpha$, then it has type $\tau$, where $\alpha$ is "fresh"...

Why is this broken?

We could immediately *universally* quantify over $\alpha$, 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 $\alpha$.

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* $\alpha, x$ = *unpack* $M_1$ *in* $x$", which would be equivalent to the unsound "*unpack* $M$" of the previous slide.

The fact that $\alpha$ is bound within $M_2$ forces it to be treated abstractly.

In fact, $M_2$ must be ??? in $\alpha$.

## On existential elimination

In fact, $M_2$ must be *polymorphic* in $\alpha$: the second premise could be:

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

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

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

One could even view "$\textit{unpack}_{\exists\alpha.\tau_1}$" as a *constant* with all these types:

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

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

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

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

## On existential introduction

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

If desired, "$\textit{pack}_{\exists\alpha.\tau}$" could also be viewed as a *constant* with all the types:

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

*i.e.* with polymorphic type:

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

## Existentials as constants

In System F, existential types can also be presented as 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* $\alpha$, if you have a $\tau$, then, for *some* $\alpha$, you have a $\tau$;
- if, for *some* $\alpha$, you have a $\tau$, then, (for any $\beta$,) if you wish to obtain a $\beta$ out of it, you must present a function which, for *any* $\alpha$, obtains a $\beta$ out of a $\tau$.

This is somewhat reminiscent of ordinary first-order logic:
$\exists x.F$ is equivalent to, and can be defined as, $\neg(\forall x.\,\neg F)$.

Is there an encoding of existential types into universal types?

## Encoding existentials into universals

The type translation is *double negation:*

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

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

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

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

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

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

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:

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

We add the reduction rule:

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

### Exercise

*Show that subject reduction and progress hold.*

## The semantics of existential types beware!

The reduction rule for existentials destructs its arguments.

Hence, *let* $\alpha, x$ = *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* $\alpha, x$ = *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* $\tau_i, V_i$ *as* $\exists\alpha.\tau$ and the two witnesses $\tau_1$ and $\tau_2$ differ.

There is no common type for the unpacking of the two possible results $V_1$ and $V_2$. The choice between those two possible results must be made, by evaluating $M_1$, before unpacking.

## 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 $\tau'$ annotation superfluous?*

- The type $\tau_0$ of $M$ is fully determined by $M$. Given the type $\exists\alpha.\tau$ of the packed value, checking that $\tau_0$ 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 $\tau'$. 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

## 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:

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

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 $\tau_0$ 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 $\tau_0$ with incompatible witness types, *e.g.* $\lambda f.\lambda x.1+(f\ (1+x))$ and $\lambda f.\lambda x.\mathit{not}\ (f\ (\mathit{not}\ x))$.

Let $v$ be the function $\lambda b.$ if $b$ then $v_1$ else $v_2$ of type *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 $\varnothing\not\vdash a_2:\tau$.

What happened? The term $a_1$ is well-typed since $v\ v_0$ has type $\tau_0$, hence $x$ can be assumed of type $(\beta\to\beta)\to(\beta\to\beta)$ for some unknown type $\beta$ and $\lambda y.y$ is of type $\beta\to\beta$.

However, without the outer existential type $v\ v_0$ can only be typed with $(\forall\alpha.\alpha\to\alpha)\to\exists\alpha.(\alpha\to\alpha)$, because the value returned by the function need different witnesses for $\alpha$. This is demanding too much on its argument and the outer application is ill-typed.

## 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 break.

## 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 $(4)$:

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

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

## 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 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:*

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

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 iso-recursive 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:

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

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 $\textsc{Gen}$ can prove.

## Iso-existential types in ML

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

The constant $\textit{pack}_D$ admits an ML type scheme, so it is unproblematic.

The construct $\textit{unpack}_D$ leads to this constraint generation rule (see type inference):

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

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

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

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:

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

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

## An example

Define

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

A client that regards $\beta$ 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 We reuse the type $D\ \alpha \approx \exists\beta.(\beta \to \alpha) \times \beta$ of frozen computations. Assume given a list $l$ with elements of type $D\ \tau_1$.

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 $D\ \tau_2$ (without actually running any computation).

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

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 D(f, y) = z in D((λ(z) g (f z)), y))
```

In expression *let* $\alpha, x$ = *unpack* $M_1$ *in* $M_2$, occurrences of $x$ in $M_2$ can only be passed to external functions (free variables) that are polymorphic so that $x$ 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 Thunk (succ, 1) where Thunk is either one of

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

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

Via first-class modules (abstract types)

```
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
```

## Contents

## An introduction to GADTs

## 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 call site

```
type ('a, 'b) apply =
 | Fadd  : (int * int, int) apply
 | Fnot  : (bool, bool) apply
 | Fstep : int → (int, int) apply
 | Fbody : (bool, int) apply
```

**Key point** the typechecker refines the types a and b in each branch

```
let rec apply : type a b. (a, b) apply → a → b =  fun f arg →
  match f with                                          (* a           b    *)
  | Fadd    → let x, y = arg in x + y                   (* int * int   int  *)
  | Fnot    → let x = arg in if x then false else true  (* bool        bool *)
  | Fstep y → let x = arg in apply Fadd (x, y)          (* int         int  *)
  | Fbody   → let b = arg in                            (* bool        int  *)
              let step = Fstep (if not b then 1 else 2)
              in apply step (apply step 0)
in apply Fbody true
```

## Example Typed evaluator

A typed abstract syntax tree

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

### Exercise

Define a typed abstract syntax tree for the simply-typed lambda-calculus and a typed evaluator.

## Example Encoding sum types

```
type ('a, 'b) sum = Left of 'a | Right of 'b
```

can be encoded as a product:

```
type ('t, 'a, 'b) tag = Ltag : ('a, 'a, 'b) tag | Rtag : ('b, 'a, 'b) tag
type ('a, 'b) prod = Prod : ('t, 'a, 'b) tag * 't → ('a, 'b) 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 constructor and need not be allocated.

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

```
type ('a, 'b) prod = Prod : ('t, 'a, 'b) tag * 't * bool → ('a, 'b) prod
let get (type a b) (p : (a, b) prod) : bool =
  let Prod (t, v, s) = p in s
```

### Exercise

*Can we have a flat representation if 'a is int * int and 'b is bool?*

## Example Encoding sum types

### Exercise

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

## Example Generic programming

```
type 'a ty =
  | Tint : int ty
  | Tbool : bool ty
  | Tlist : 'a ty → ('a list) ty
  | Tpair : 'a ty ∗ 'b ty → ('a ∗ 'b) 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)
```

## Other uses of GADTs

### GADTs

- May encode data structures 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 (version inspired by the generic printer)
- GADTs can be used to encode type classes, using a technique analogous to defunctionalization [Pottier and Gauthier, 2006].

## Reducing GADTs to type equality

All GADTs can be encoded with a single one:

```
type ('a, 'b) eq = Eq : ('a, 'a) eq
```

For instance, generic programming can be redefined as follows:

```
type 'a ty =
  | Tint  : ('a, int) eq → 'a ty
  | Tlist : ('a, 'b list) eq * 'b ty → 'a ty
  | Tpair : ('a, ('b * 'c)) eq * 'b ty * 'c ty → 'a 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, t) → "[" ^ String.concat "; " (List.map (to_string t) 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

All GADTs can be encoded with a single one:

```
type ('a, 'b) eq = Eq : ('a, 'a) eq
```

For instance, generic programming can be redefined as follows:

```
type 'a ty =
  | Tint  : ('a, int) eq → 'a ty
  | Tlist : ('a, 'b list) eq * 'b ty → 'a ty
  | Tpair : ('a, ('b * 'c)) eq * 'b ty * 'c ty → 'a 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, t) → ...
 | Tpair (Eq, a, b) → ...
```

▷ Pattern "Tint *Eq*" is GADT matching

## Reducing GADTs to type equality

All GADTs can be encoded with a single one:

```
type ('a, 'b) eq = Eq : ('a, 'a) eq
```

For instance, generic programming can be redefined as follows:

```
type 'a ty =
  | Tint  : ('a, int) eq → 'a ty
  | Tlist : ('a, 'b list) eq * 'b ty → 'a ty
  | Tpair : ('a, ('b * 'c)) eq * 'b ty * 'c ty → 'a 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 p = Eq in string_of_int x
 | Tlist (Eq, t) → ...
 | Tpair (Eq, a, b) → ...
```

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

## Formalisation of GADTs

We can encode GADTs with type equalities

We cannot encode type equalities in System F.

They bring something more, namely *local equalities* in the typing context.

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

When typechecking an expression

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

- $M$ is typechecked with the asumption that $\tau_1 \sim \tau_2$, *i.e.* types $\tau_1$ and $\tau_2$ 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

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

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

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

## Fc (simplified) *Conversion*

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

## 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:

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

## 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* $(\mathit{bool} \triangleleft 0)$, which is well-typed in this context.

▷ In call-by-value,

| | | |
|---|---|---|
| $\lambda x : \tau_1 \sim \tau_2 .\, M$ | freezes | the evaluation of $M$, |
| $M \triangleleft \gamma$ | 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 non obvious.

$$\begin{array}{lll} \gamma \triangleleft M & \text{erases} & \text{to } M \\ \gamma & \text{erases} & \text{to } \diamond \end{array}$$

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

$$\textsc{Coerce}\ \frac{\Gamma \vdash M : \tau_1 \qquad \Gamma \Vdash \tau_1 \sim \tau_2}{\Gamma \vdash M : \tau_2} \qquad \textsc{Coercion}\ \frac{\Gamma \Vdash \tau_1 \sim \tau_2}{\Gamma \vdash \diamond : \tau_1 \sim \tau_2} \qquad \textsc{CoAbs}\ \frac{\Gamma, x : \tau_1 \sim \tau_2 \vdash M : \tau}{\Gamma \vdash \lambda x : \tau_1 \sim \tau_2 . M : \tau}$$

## 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 present of possibly uninhabited coercions.

## Type soundness *Semantic proofs*

Equality coercions are a small logic of type conversions.

This may be enriched with more operations.

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

The 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 the return type annotion 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.

## 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.

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.

▷ 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.

## Bibliography IV

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.

## Bibliography VI

▷ 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.

▷ François Pottier and Yann Régis-Gianas. Towards efficient, typed LR parsers. In *ACM Workshop on ML*, volume 148 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.

## Bibliography VII

▷ 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.

▷ 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.

## Bibliography VIII

▷ 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.

▷ 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.