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

Didier Rémy

September 29, 2021

Plan of the course

Metatheory of System F

ADTs, Recursive types, Existential types, GATDs

Going higher order with F^ω!

Logical relations

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):

?

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,

?

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 $\quad C_i \; : \; \forall \vec{\alpha}.\, \tau_i \to G\,\vec{\alpha}$

?

Algebraic Datatypes Types — General case

General case

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

In System F, this amounts to declaring (implicit version for conciseness):

- a new type constructor G,
- n constructors $\quad C_i : \forall\vec{\alpha}.\,\tau_i \to G\,\vec{\alpha}$
- one destructor $\quad d_G : \forall\vec{\alpha},\gamma.\,G\,\vec{\alpha} \to (\tau_1 \to \gamma)\ldots(\tau_n \to \gamma) \to \gamma$

?

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 $\quad C_i \;:\; \forall\vec{\alpha}.\,\tau_i \to G\,\vec{\alpha}$
- one destructor $\quad d_G \;:\; \forall\vec{\alpha},\gamma.\,G\,\vec{\alpha} \to (\tau_1 \to \gamma)\ldots(\tau_n \to \gamma) \to \gamma$
- n reduction rules $\quad d_G\,(C_i\;v)\;v_1\;\ldots v_n \;\rightsquigarrow\; v_i\;v$

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

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:

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

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

Equi- versus 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:

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

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

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 — Equality

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.

Equirecursive types

Equality

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

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

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

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 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.)

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 $\mu\alpha A A A\alpha$

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

Equirecursive types — Without quantifiers

Proof of $\mu\alpha AA\alpha = \mu\alpha AAA\alpha$

By coinduction

By unification

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

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 gray, 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

Type soundness

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

?

Equirecursive types

Type soundness

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 was is 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)?

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 *rec* $\alpha.\alpha \to \alpha$
- 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 iso- and) equirecursive types.

- equirecursive types are restricted by default to object or data types.
- 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:

?

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:

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

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?

?

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,

?

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* $C_\Pi \;:\; \forall \vec{\alpha}.\, \tau_1 \to \dots \tau_n \to G\,\vec{\alpha}$

?

Records

A record can be defined as

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

Exercise

What are the corresponding declarations in System F?

- *a new type constructor* G_Π,
- 1 *constructor* $C_\Pi \;:\; \forall \vec{\alpha}.\, \tau_1 \to \dots \tau_n \to G\,\vec{\alpha}$
- n *destructors* $d_{\ell_i} \;:\; \forall \vec{\alpha}.\, G\,\vec{\alpha} \to \tau_i$

?

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* $C_\Pi \;:\; \forall \vec{\alpha}.\, \tau_1 \to \ldots \tau_n \to G\,\vec{\alpha}$
- n *destructors* $d_{\ell_i} \;:\; \forall \vec{\alpha}.\, G\,\vec{\alpha} \to \tau_i$
- n *reduction rules* $d_{\ell_i}(C_\Pi\; v_1\; \ldots v_n) \;\leadsto\; v_i$

?

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* $\quad d_{\ell_i}(C_\Pi\, v_1\, \ldots v_n) \;\rightsquigarrow\; v_i$

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]

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 with GADTs.)

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

Contents

Existential types *Examples*

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

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

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:

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

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:

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

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:

Pack

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

Unpack

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

Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

Pack

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

Unpack

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

Anything wrong?

Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

Pack

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

Unpack

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

The 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 $\alpha \notin \mathrm{dom}(\Gamma)$.

Existential types in explicit style

Here is how the existential quantifier is introduced and eliminated:

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

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

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:

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

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?

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 "$\textit{let}\ \alpha, x = \textit{unpack}\ M_1\ \textit{in}\ x$", which would be equivalent to the unsound "$\textit{unpack}\ M$" 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:

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

On existential elimination

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

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

On existential elimination

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

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

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

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

On existential elimination

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

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

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 family of *constants* of types:

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

On existential elimination

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

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

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 family of *constants* of types:

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

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

On existential elimination

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

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

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 family of *constants* of types:

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

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 $\alpha \# \tau_2$.

On existential introduction

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

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:*

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

The term translation is:

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

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:

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

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:

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

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:

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

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:

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

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:

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

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

What is the computational content of this encoding?

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:

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

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:

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

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.

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

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.*

Solution

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

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

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:

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

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:

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

Is the semantics type-erasing?

Implicitly-typed existential types subtlety

Yes, it is.

But there is a subtlety!

Implicitly-typed existential types subtlety

Yes, it is.

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

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 $bool \to \tau_0$.

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

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

What happened?

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 $\lambda f.\lambda x.\mathit{not}\ (f\ (\mathit{not}\ x))$.

Let v be the function $\lambda b.$ if b then v_1 else v_2 of type *bool* $\to \tau_0$.

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

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

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:

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

Comments?

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:

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

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* 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?

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:

UNPACK$_D$

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

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 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:

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

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:

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

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

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).

$$\mathsf{List.map}\ (\lambda(\mathsf{z})\ \textbf{let}\ \mathsf{D(f, y) = z}\ \textbf{in}\ \mathsf{D((\lambda(z)\ g\ (f\ z)), y)})$$

Try generalizing this example to a function that receives g and l and returns l'

?

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))
```

?

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* 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 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 $\exists \beta (\beta \to int) \times \beta$ 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:

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

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 encoding 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 d = D : ('b → 'a) * 'b → 'a d
let freeze f x = D (f, x)
let unfreeze (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

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

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

What are they?

ADTs

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

```
type α list =
  | Nil : α list
  | Const : α * α list → α 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
```

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
```

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 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
```

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 b → let x = arg in                                (* int        int   *)
              apply Fadd (x, if apply Fnot b then 1 else 2)
  | Fbody   → let b = arg in                                (* bool       int   *)
              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 ∗ 'a expr ∗ 'a expr) → 'a expr
let e0         = (If (Zerop (Int 0), Int 1, Int 2))
```

What is the type of e0?

Examples

Typed evaluator

A typed abstract-syntax tree

```
type _ 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                                    (∗ 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
```

Examples Typed evaluator

A typed abstract-syntax tree

```
type _ 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                                   (* 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

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

Examples

Generic programming

Example of printing

```
type _ 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)
```

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:

$$\textbf{type}\ (\alpha, \beta)\ \text{eq} = \textit{Eq} : (\alpha, \alpha)\ \text{eq}$$

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 possibkle uses* of the constructors.

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!

```
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 p” is ordinary ADT matching
- ▷ **let** *Eq* = p **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 (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

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

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

Typing rules:

COERCE

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

COERCION

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

COABS

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

Coercions witness type equivalences:

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

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

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

?

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

Always?

Semantics

Coercions should be without computational content

Except ...

?

Semantics

Coercions should be without computational content

Except for coercion abstractions that must stop the evaluation

Why?

Semantics

Coercions should be without computational content

Except for coercion abstractions that must stop the evaluation

- Otherwise, one could attempt to reduce M in $\lambda \textit{int} \sim \textit{bool}.\, M$ when M is *not* $(\textit{bool} \triangleleft 0)$, 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*

?

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,

?

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 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.

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

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

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

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.