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

Didier Rémy

October 28, 2020

INRIA
Plan of the course

Metatheory of System F

ADTs, Existential types, GATDs
Abstract Data types, Existential types, GADTs
# Contents

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

Examples

In OCaml:

```ocaml
letrec
  type 'a list =
  | Nil : 'a list
  | Cons : 'a * 'a list -> 'a list

or

letrec
  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

\[
\text{type } G \bar{\alpha} = \sum_{i \in 1 .. n} (C_i : \forall \bar{\alpha} . \tau_i \rightarrow G \bar{\alpha}) \quad \text{where } \bar{\alpha} = \bigcup_{i \in 1 .. n} \text{ftv}(\tau_i)
\]

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

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

Exercise

_Show that this extension verifies the subject reduction and progress axioms for constants._
Algebraic Datatypes Types

**General case**

\[
\text{type } G \vec{\alpha} = \sum_{i \in 1..n} (C_i : \forall \vec{\alpha}. \tau_i \rightarrow G \vec{\alpha}) \quad \text{where } \vec{\alpha} = \bigcup_{i \in 1..n} \text{ftv}(\tau_i)
\]

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

Typed closure conversion

Existential types

- Implicitly-type existential types passing
- Iso-existential types

Typed closure conversion

- Environment passing
- Closure passing

Generalized Algebraic Datatypes
Recursive Types

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

Indeed, if the grammar of types is $\tau ::= \text{unit} \mid \tau \times \tau \mid \tau + \tau$, then it is clear that every type describes a \textit{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:

$$\text{list}\;\alpha \diamond \;\text{unit} + \alpha \times \text{list}\;\alpha$$

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:

\[ \tau ::= \alpha \mid F \bar{\tau} \mid \forall \beta. \tau \]

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:

\[ \tau ::= \alpha \mid \mu\alpha.(F \tilde{\tau}) \mid \mu\alpha.(\forall\beta.\tau) \]

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:

\[ \mu\beta.(\text{unit} + \alpha \times \beta) \]
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 \textit{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 \textit{decide} whether two types are \textit{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:

\[
\begin{align*}
\text{Fold/Unfold} & \quad \mu \alpha. \tau = [\alpha \mapsto \mu \alpha. \tau] \tau \\
\text{Uniqueness} & \quad \tau_1 = [\alpha \mapsto \tau_1] \tau \\
& \quad \tau_2 = [\alpha \mapsto \tau_2] \tau \\
& \quad \tau_1 = \tau_2
\end{align*}
\]

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:

\[
\begin{align*}
\alpha & = \alpha \\
[\alpha \mapsto \mu \alpha. F \vec{\tau}] \vec{\tau} & = [\alpha \mapsto \mu \alpha. F \vec{\tau}'] \vec{\tau}' \\
\mu \alpha. F \vec{\tau} & = \mu \alpha. F \vec{\tau}' \\
[\alpha \mapsto \mu \alpha. \forall \beta. \tau] \tau & = [\alpha \mapsto \mu \alpha. \forall \beta. \tau'] \tau' \\
\mu \alpha. \forall \beta. \tau & = \mu \alpha. \forall \beta. \tau'
\end{align*}
\]

Exercise

Show that \( \mu \alpha. A \alpha = \mu \alpha. A A \alpha \) and \( \mu \alpha. A B \alpha = A \mu \alpha. B A \alpha \) 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 $F \tilde{\tau}_1 = F \tilde{\tau}_2$ implies $\tilde{\tau}_1 = \tilde{\tau}_2$—this was used in the proof of Subject Reduction.

It remains true that $F_1 \tilde{\tau}_1 = F_2 \tilde{\tau}_2$ implies $F_1 = F_2$—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:

$$\text{type } G \tilde{\alpha} = \Sigma(C : \forall \tilde{\alpha}. [\beta \mapsto G \tilde{\alpha}] \tau \to G \tilde{\alpha})$$

where $\tilde{\alpha} = \text{ftv}(\tau) \setminus \{\beta\}$

The constructor $C$ coerces $[\beta \mapsto G \tilde{\alpha}] \tau$ to $G \tilde{\alpha}$ 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

\[ \text{type } G \bar{\alpha} = \Pi_{i \in 1..n}(\ell_i : \tau_i) \]

where \( \bar{\alpha} = \bigcup_{i \in 1..n} \text{ftv}(\tau_i) \)

Exercise

*What are the corresponding declarations in System F?*

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

*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]
Regular ADTs

type $G \tilde{\alpha} = \Sigma_{i \in 1..n}(C_i : \forall \tilde{\alpha}. \tau_i \rightarrow G \tilde{\alpha})$

If all occurrences of $G$ in $\tau_i$ are $G \tilde{\alpha}$ 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

- **Algebraic Data Types**
  - Equi- and iso-recursive types

- **Typed closure conversion**

- **Existential types**
  - Implicitly-type existential types passing
  - Iso-existential types

- **Typed closure conversion**
  - Environment passing
  - Closure passing

- **Generalized Algebraic Datatypes**
Type-preserving compilation

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

Why *preserve types* during compilation?

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

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

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

For example:

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

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

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

In general, a type-preserving compilation phase involves not only a translation of *terms*, mapping $M$ to $\llbracket M \rrbracket$, but also a translation of *types*, mapping $\tau$ to $\llbracket \tau \rrbracket$, with the property:

$$\Gamma \vdash M : \tau \quad \text{implies} \quad \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket \tau \rrbracket$$

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

See the old lecture on type closure conversion.
Closure conversion

First-class functions may appear in the body of other functions. hence, their own body may contain free variables that will be bound to values during the evaluation in the execution environment.

Because they can be returned as values, and thus used outside of their definition environment, they must store their execution environment in their value.

A closure is the packaging of the code of a first-class function with its runtime environment, so that it becomes closed, i.e. independent of the runtime environment and can be moved and applied in another runtime environment.

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

In the following,

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

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

There are at least two variants of closure conversion:

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

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

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

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

\[
\begin{align*}
\llbracket \lambda x. a \rrbracket &= \text{let } \text{code} = \lambda (\text{clo}, x). \\
&\quad \text{let } (\_, x_1, \ldots, x_n) = \text{clo } \text{in } \llbracket a \rrbracket \text{ in } \\
&\quad (\text{code}, x_1, \ldots, x_n) \\
\end{align*}
\]

\[
\begin{align*}
\llbracket a_1 \ a_2 \rrbracket &= \text{let } \text{clo} = \llbracket a_1 \rrbracket \text{ in } \\
&\quad \text{let } \text{code} = \text{proj}_0 \ \text{clo } \text{in } \\
&\quad \text{code } (\text{clo}, \llbracket a_2 \rrbracket)
\end{align*}
\]

(The variables \text{code} and \text{clo} must be suitably fresh.)

**Important!** The layout of the environment must be known only at the closure allocation site, not at the call site. In particular, \( \text{proj}_0 \ \text{clo} \) need not know the size of \( \text{clo} \).
Environment-passing closure conversion

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

\[
\begin{align*}
[\lambda x. a] &= \text{let } \text{code} = \lambda (env, x). \\
&\quad \text{let } (x_1, \ldots, x_n) = env \text{ in } [a] \text{ in } \\
&\quad (\text{code}, (x_1, \ldots, x_n))
\end{align*}
\]

\[
[\lambda_1 \lambda_2] = \text{let } (\text{code}, env) = [\lambda_1] \text{ in } \\
\text{code} (env, [\lambda_2])
\]

Questions: How can closure conversion be made type-preserving?

The key issue is to find a sensible definition of the type translation. In particular, what is the translation of a function type, \([\tau_1 \rightarrow \tau_2]\)?
Environment-passing closure conversion

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

\[
\left[\lambda x. a\right] = \text{let code} = \lambda (\text{env}, x).
\text{let } (x_1, \ldots, x_n) = \text{env in } [a] \text{ in }
\left(\text{code}, (x_1, \ldots, x_n)\right)
\]

Assume \( \Gamma \vdash \lambda x. a : \tau_1 \to \tau_2 \).

Assume, \textit{w.l.o.g.} \( \text{dom}(\Gamma) = \text{fv}(\lambda x. a) = \{x_1, \ldots, x_n\} \).

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

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

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

Now, \textit{what should be the definition of }\([\tau_1 \to \tau_2]\)?
Towards a type translation

Can we adopt this as a definition?

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

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

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

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

Yes, we do.

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

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

\[
\text{if } \ldots \text{ then } \lambda x. x + y \text{ else } \lambda x. x
\]

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

So, *what could be the definition of } \llbracket \tau_1 \to \tau_2 \rrbracket *?
The type translation

The only sensible solution is:

\[
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \exists \alpha.((\alpha \times \llbracket \tau_1 \rrbracket) \rightarrow \llbracket \tau_2 \rrbracket) \times \alpha
\]

An *existential quantification* over the type of the environment abstracts away the differences in size and layout.

Enough information is retained to ensure that the application of the code to the environment is valid: this is expressed by letting the variable \( \alpha \) occur twice on the right-hand side.
The type translation

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

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

\[
\text{let } f = \text{let } x = \text{ref } 0 \text{ in } \lambda(). x := (x + 2); ! x
\]

After closure conversion, the reference \( x \) is reachable via the closure of \( f \). A malicious, untyped client could write an odd value to \( x \). However, a well-typed client is unable to do so.

This encoding is not just type-preserving, but also fully abstract: it preserves (a typed version of) observational equivalence [Ahmed and Blume, 2008].
Contents

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

A frozen application returning a value of type \((\approx \text{a thunk})\)
\[\exists \alpha. (\alpha \to \tau) \times \alpha\]

Type of closures in the environment-passing variant:
\[\llbracket \tau_1 \to \tau_2 \rrbracket = \exists \alpha. ((\alpha \times \llbracket \tau_1 \rrbracket) \to \llbracket \tau_2 \rrbracket) \times \alpha\]

A possible encoding of objects:
\[
= \exists \rho.
\begin{align*}
&\mu \alpha. \\
&\Pi \left( \begin{array}{l}
(\alpha \times \tau_1) \to \tau_1' ; \\
\ldots
\end{array} \right ; \\
&\left( \begin{array}{l}
(\alpha \times \tau_n) \to \tau_n' \\
\rho
\end{array} \right)
\end{align*}
\]
\[
\rho \text{ describes the state} \\
\alpha \text{ is the concrete type of the closure} \\
a \text{tuple...} \\
\ldots \text{that begins with a record...} \\
\ldots \text{of method code pointers...} \\
\ldots \text{and continues with the state} \\
(a \text{tuple of unknown length})
\]
Existential types

One can extend System F with *existential types*, in addition to universals:

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

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**

\[
\Gamma \vdash M : [\alpha \mapsto \tau']\tau
\]

\[
\Gamma \vdash pack\tau', M \text{ as } \exists \alpha. \tau : \exists \alpha. \tau
\]

**Unpack**

\[
\Gamma \vdash M_1 : \exists \alpha. \tau_1
\]

\[
\Gamma, \alpha, x : \tau_1 \vdash M_2 : \tau_2 \quad \alpha \not\equiv \tau_2
\]

\[
\Gamma \vdash let \alpha, x = unpack M_1 \text{ in } M_2 : \tau_2
\]

Anything wrong? The side condition \(\alpha \not\equiv \tau_2\) is mandatory here to ensure well-formedness of the conclusion.

The side condition may also be written \(\Gamma \vdash \tau_2\) which implies \(\alpha \not\equiv \tau_2\), given that the well-formedness of the last premise implies \(\alpha \not\in \text{dom}(\Gamma)\).

Note the *imperfect duality* between universals and existentials:

**T\text{Abs}**

\[
\Gamma, \alpha \vdash M : \tau
\]

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

**T\text{App}**

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

\[
\Gamma \vdash M \tau' : [\alpha \mapsto \tau']\tau
\]
On existential elimination

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

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

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

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 \( \Gamma \vdash \Lambda \alpha. \text{unpack} M : \forall \alpha. \tau \). This is nonsense!

Replacing the premise \( \Gamma, \alpha \vdash M : \exists \alpha. \tau \) by the conjunction \( \Gamma \vdash M : \exists \alpha. \tau \) and \( \alpha \in \text{dom}(\Gamma) \) 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 \( M_2 \).

\[
\text{UNPACK} \\
\Gamma \vdash M_1 : \exists \alpha. \tau_1 \\
\Gamma, \alpha; x : \tau_1 \vdash M_2 : \tau_2 \quad \alpha \not\equiv \tau_2 \\
\Gamma \vdash \text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2 : \tau_2
\]

The restriction \( \alpha \not\equiv \tau_2 \) prevents writing "let \( \alpha, x = \text{unpack } M_1 \text{ in } x \)" which would be equivalent to the unsound "\( \text{unpack } M \)" 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:

$$
\begin{align*}
\Gamma \vdash M_1 : \exists \alpha. \tau_1  \\
\Gamma \vdash \Lambda \alpha. \lambda x : \tau_1. M_2 : \forall \alpha. \tau_1 \to \tau_2  \\
\alpha \not\equiv \tau_2
\end{align*}
$$

or let $\alpha, x = \text{unpack } M_1 \text{ in } M_2 : \tau_2$

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

$$
\begin{align*}
\Gamma \vdash M_1 : \exists \alpha. \tau_1  \\
\Gamma \vdash N_2 : \forall \alpha. \tau_1 \to \tau_2  \\
\alpha \not\equiv \tau_2
\end{align*}
$$

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

$$
\text{unpack}_{\exists \alpha. \tau_1} : (\exists \alpha. \tau_1) \to (\forall \alpha. (\tau_1 \to \tau_2)) \to \tau_2  \\
\alpha \not\equiv \tau_2
$$

Thus,

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

or, better

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

$\beta$ 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 $\alpha \not\equiv \tau_2$. 
On existential introduction

\[
\text{PACK} \quad \frac{\Gamma \vdash M : [\alpha \mapsto \tau'] \tau}{\Gamma \vdash \text{pack } \tau', M \text{ as } \exists \alpha. \tau : \exists \alpha. \tau}
\]

If desired, “\(\text{pack}_{\exists \alpha. \tau}\)” could also be viewed as a \textit{constant} with all the types:

\[
\text{pack}_{\exists \alpha. \tau} : \quad [\alpha \mapsto \tau'] \tau \rightarrow \exists \alpha. \tau
\]

\textit{i.e.} with polymorphic type:

\[
\text{pack}_{\exists \alpha. \tau} : \quad \forall \alpha. (\tau \rightarrow \exists \alpha. \tau)
\]
Existentials as constants

In System F, existential types can also be presented as constants

\[
\text{pack}_{\exists \alpha. \tau} : \forall \alpha. (\tau \rightarrow \exists \alpha. \tau)
\]

\[
\text{unpack}_{\exists \alpha. \tau} : \exists \alpha. \tau \rightarrow \forall \beta. ((\forall \alpha. (\tau \rightarrow \beta)) \rightarrow \beta)
\]

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*:

$$\left[\exists \alpha. \tau\right] = \forall \beta. (\forall \alpha. (\left[\tau\right] \rightarrow \beta)) \rightarrow \beta \quad \text{if } \beta \neq \tau$$

The term translation is:

$$\left[\text{pack}_{\exists \alpha. \tau}\right] : \forall \alpha. (\left[\tau\right] \rightarrow \left[\exists \alpha. \tau\right])$$

$$= \Lambda \alpha. \lambda x: [\tau]. \Lambda \beta. \lambda k: \forall \alpha. (\left[\tau\right] \rightarrow \beta). k \alpha x$$

$$\left[\text{unpack}_{\exists \alpha. \tau}\right] : \left[\exists \alpha. \tau\right] \rightarrow \forall \beta. (\forall \alpha. (\left[\tau\right] \rightarrow \beta)) \rightarrow \beta$$

$$= \lambda x: [\exists \alpha. \tau]. x$$

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

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

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

It would be more intuitive, however, to treat \( \text{unpack} \exists \alpha. \tau_0 \) as a binary destructor:

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

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

We extend values and evaluation contexts as follows:

\[
\begin{align*}
V & ::= \ldots \text{pack } \tau', V \text{ as } \tau \\
E & ::= \ldots \text{pack } \tau', [] \text{ as } \tau \mid \text{let } \alpha, x = \text{unpack } [] \text{ in } M
\end{align*}
\]

We add the reduction rule:

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

Exercise

Show that subject reduction and progress hold.
The semantics of existential types

The reduction rule for existentials destructs its arguments.

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

beware!
The semantics of existential types

**Exercise**

*Find an example that illustrates why the reduction of let $\alpha, x = \text{unpack } M_1 \text{ in } M_2$ 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 $\text{pack } \tau_i, V_i$ 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:*

\[
\Gamma \vdash M : [\alpha \mapsto \tau'] \tau
\]

\[
\Gamma \vdash \text{pack } \tau', M \text{ as } \exists \alpha. \tau : \exists \alpha. \tau
\]

*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
- Typed closure conversion
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Typed closure conversion
  - Environment passing
  - Closure passing
- 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:

\[
\text{Unpack} \quad \frac{\Gamma \vdash a_1 : \exists \alpha. \tau_1 \quad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \quad \alpha \neq \tau_2}{\Gamma \vdash \text{let } x = a_1 \text{ in } a_2 : \tau_2}
\]

\[
\text{Pack} \quad \frac{\Gamma \vdash a : [\alpha \mapsto \tau'] \tau}{\Gamma \vdash a : \exists \alpha. \tau}
\]

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:

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

Is the semantics type-erasing?
Implicitly-typed existential types

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:

\[
\text{let } x = M_1 \text{ in } M_2 \rightarrow [x \mapsto M_1]M_2
\]

With existential types, this breaks subject reduction!

Why?
Implicitly-typed existential types

Let $\tau_0$ be $\exists \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha)$ and $v_0$ a value of type $\text{bool}$. 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 $\lambda f. \lambda x. \text{not} (f (\text{not} x))$.

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

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

We have $\emptyset \vdash a_1 : \exists \alpha. \alpha \to \alpha$ while $\emptyset \not\vdash a_2 : \tau$.

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.
Impliclty-typed existential types

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:

\[
\Gamma \vdash a_1 : \exists \alpha.\tau_1 \quad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \quad \alpha \not\equiv \tau_2
\]

\[
\Gamma \vdash [x \mapsto a_1]a_2 : \tau_2
\]

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

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

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):

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

In the call-by-value setting, \( \lambda k. \llbracket a \rrbracket k \) would come from the reduction of \( \llbracket \text{pack } a \rrbracket \), i.e. is \( (\lambda k. \lambda x. k \ x) \llbracket a \rrbracket \), so that \( a \) is always a value \( v \).

However, \( a \) need not be a value. What is essential is that \( a_1 \) be reduced to some head normal form \( \lambda k. \llbracket a \rrbracket k \).
• Algebraic Data Types
  • Equi- and iso-recursive types

• Typed closure conversion

• Existential types
  • Implicitly-type existential types passing
  • Iso-existential types

• Typed closure conversion
  • Environment passing
  • Closure passing

• 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*:

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

This introduces two constants, with the following type schemes:

\[ \text{pack}_D : \forall \tilde{\alpha} \tilde{\beta}. \tau \rightarrow D \tilde{\alpha} \]
\[ \text{unpack}_D : \forall \tilde{\alpha} \gamma. D \tilde{\alpha} \rightarrow (\forall \tilde{\beta}. (\tau \rightarrow \gamma)) \rightarrow \gamma \]

(Compare with basic iso-recursive types, where \( \tilde{\beta} = \emptyset \).)
Iso-existential types in ML

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

$$\forall \bar{\alpha} \gamma. D \bar{\alpha} \rightarrow (\forall \bar{\beta}. (\tau \rightarrow \gamma)) \rightarrow \gamma$$

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

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

\[
\text{Unpack}_D
\]

\[
\begin{align*}
\Gamma & \vdash M_1 : D \bar{\bar{\tau}} \\
\Gamma & \vdash M_2 : \forall \bar{\beta}. ([\bar{\alpha} \mapsto \bar{\bar{\tau}}] \tau \rightarrow \tau_2) \quad \bar{\beta} \neq \bar{\bar{\tau}}, \tau_2 \\
\Gamma & \vdash \text{unpack}_D M_1 M_2 : \tau_2
\end{align*}
\]

where $D \bar{\alpha} \approx \exists \bar{\beta}. \tau$

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 $\text{GEN}$ can prove.
Iso-existential types in ML

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

The constant \( \text{pack}_D \) admits an ML type scheme, so it is unproblematic.

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

\[
\llbracket \text{unpack}_D \ M_1 \ M_2 : \tau_2 \rrbracket = \exists \bar{\alpha}. \left( \llbracket M_1 : D \bar{\alpha} \rrbracket \ \forall \bar{\beta}. \llbracket M_2 : \tau \rightarrow \tau_2 \rrbracket \right)
\]

where \( D \bar{\alpha} \approx \exists \bar{\beta}.\tau \) and, \( \text{w.l.o.g., } \bar{\alpha} \bar{\beta} \not\equiv M_1, M_2, \tau_2. \)

A universally quantified constraint appears where polymorphism is \textit{required}. 
Iso-existential types in ML

In practice, Läuffer 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:

\[
\text{type } D \bar{\alpha} = \ell : \tau \rightarrow D \bar{\alpha}
\]

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

\[
\langle \text{match } M_1 \text{ with } \ell x \rightarrow M_2 : \tau_2 \rangle = \exists \bar{\alpha}. \left( \langle M_1 : D \bar{\alpha} \rangle \ \forall \bar{\beta}. \text{def } x : \tau \text{ in } \langle M_2 : \tau_2 \rangle \right)
\]

where, w.l.o.g., \(\bar{\alpha} \bar{\beta} \not\# M_1, M_2, \tau_2\).
An example

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

$$\langle \text{unpack}_{\text{Any}} M_1 (\lambda x. x) : \tau_2 \rangle = \langle M_1 : \text{Any} \rangle \land \forall \beta. \langle \lambda x. x : \beta \rightarrow \tau_2 \rangle$$

$$\models \forall \beta. \beta = \tau_2$$

$$\equiv \text{false}$$

(Recall that $\beta \not\equiv \tau_2$.)


An example

Define

\[ D \alpha \approx \exists \beta. (\beta \to \alpha) \times \beta \]

A client that regards \( \beta \) as abstract succeeds:

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

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

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

$$\text{let lift } g l = \text{List.map } (\lambda(z) \text{ let } D(f, y) = z \text{ in } D((\lambda(z) g (f z)), y))$$

In expression $\text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2$, 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 writing type annotations, but in a more rigid way
- In particular, there is no canonical way of representing them. For example, a thunk of type $\exists \beta (\beta \to \text{int}) \times \beta$ could have been defined as $\text{Thunk (succ, 1)}$ where $\text{Thunk}$ is either one of

  ```plaintext
  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 \textit{abstract types}. For instance, the type:

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

specifies an abstract implementation of integer stacks.

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

Montagu and Rémy [2009] make existential types \textit{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)

```ocaml
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)

```ocaml
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

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

• Typed closure conversion

• Existential types
  • Implicitly-type existential types passing
  • Iso-existential types

• Typed closure conversion
  • Environment passing
  • Closure passing

• Generalized Algebraic Datatypes
Typed closure conversion

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

$$\Gamma \vdash M : \tau \implies [\Gamma] \vdash [M] : [\tau]$$
Environment-passing closure conversion

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

$$\left[\lambda x: \tau_1. M\right] = \begin{array}{l}
\text{let } code : \left(\left[\Gamma\right] \times \left[\tau_1\right]\right) \rightarrow \left[\tau_2\right] = \\
\lambda (env : \left[\Gamma\right], x : \left[\tau_1\right]). \\
\text{let } (x_1, \ldots, x_n : \left[\Gamma\right]) = env \text{ in } \\
\left[M\right] \\
in \\
pack \left[\Gamma\right], (\text{code}, (x_1, \ldots, x_n)) \\
as \exists \alpha.((\alpha \times \left[\tau_1\right]) \rightarrow \left[\tau_2\right]) \times \alpha
\end{array}$$

We find $\left[\Gamma\right] \vdash \left[\lambda x: \tau_1. M\right] : \left[\tau_1 \rightarrow \tau_2\right]$, as desired.
Environment-passing closure conversion

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

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

We find $\llbracket \Gamma \rrbracket \vdash \llbracket M \ M_1 \rrbracket : \llbracket \tau_2 \rrbracket$, as desired.
Recursive functions can be translated in this way, known as the “fix-code” variant [Morrisett and Harper, 1998] (leaving out type information):

\[
\llbracket \mu f. \lambda x. M \rrbracket \quad = \quad \text{let rec code (env, x) =}
\]

\[
\quad \text{let } f = \text{pack (code, env) in}
\]

\[
\quad \text{let } (x_1, \ldots, x_n) = \text{env in}
\]

\[
\llbracket M \rrbracket \quad \text{in}
\]

\[
\quad \text{pack (code, (x_1, \ldots, x_n))}
\]

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

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

What is the weak point of this variant?

A new closure is allocated at every call.
Instead, the “fix-pack” variant [Morrisett and Harper, 1998] uses an extra field in the environment to store a back pointer to the closure:

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

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

This requires general, recursively-defined \textit{values}. Closures are now \textit{cyclic} data structures.
Here is how the ‘fix-pack’ variant is type-checked. Assume
\( \Gamma \vdash \mu f.\lambda x.M : \tau_1 \rightarrow \tau_2 \) and \( \text{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \text{fv}(\mu f.\lambda x.M) \).

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

in clo

Problem?
Environment-passing closure conversion

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

We can generalize the encoding afterwards,

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

whenever the right-hand side is well-defined.

This allows the *indirect* compilation of polymorphic recursive functions as long as the recursion is monomorphic.

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

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

The encoding is simple.

However, this requires the introduction of recursive non-functional values “let rec \( x = v \)”. While this is a useful construct, it really alters the operational semantics and requires updating the type soundness proof.
- Algebraic Data Types
  - Equi- and iso-recursive types

- Typed closure conversion

- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types

- Typed closure conversion
  - Environment passing
  - Closure passing

- Generalized Algebraic Datatypes
Closure-passing closure conversion

\[
\llbracket \lambda x. M \rrbracket = \begin{array}{l}
\quad \text{let code} = \lambda (\text{clo}, x).
\quad \text{let } (\_, x_1, \ldots, x_n) = \text{clo} \text{ in}
\quad \llbracket M \rrbracket
\quad \text{in } (\text{code}, x_1, \ldots, x_n)
\end{array}
\]

\[
\llbracket M_1 \ M_2 \rrbracket = \begin{array}{l}
\quad \text{let clo} = \llbracket M_1 \rrbracket \text{ in}
\quad \text{let code} = \text{proj}_0 \text{ clo} \text{ in}
\quad \text{code} (\text{clo}, \llbracket M_2 \rrbracket)
\end{array}
\]

There are two difficulties:

- a closure is a tuple, whose \textit{first} field should be \textit{exposed} (it is the code pointer), while the number and types of the remaining fields should be abstract;

- the first field of the closure contains a function that expects \textit{the closure itself} as its first argument.
Closure-passing closure conversion

There are two difficulties:

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

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

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

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

\[
\begin{align*}
\tau & ::= \ldots \mid \Pi R & \text{− types} \\
R & ::= \epsilon \mid (\tau; R) & \text{− rows}
\end{align*}
\]

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

Let us now introduce \textit{row variables} and allow \textit{quantification} over them:

\[
\begin{align*}
\tau & ::= \ldots \mid \Pi R \mid \forall \rho. \tau \mid \exists \rho. \tau & \text{− types} \\
R & ::= \rho \mid \epsilon \mid (\tau; R) & \text{− rows}
\end{align*}
\]

This allows reasoning about the first few fields of a tuple whose length is not known.
Typing rules for tuples

The typing rules for tuple construction and deconstruction are:

\[
\begin{align*}
\text{TUPLE} & \quad \forall i. \in [1, n] \quad \Gamma \vdash M_i : \tau_i \\
\Gamma & \vdash (M_1, \ldots, M_n) : \Pi (\tau_1; \ldots; \tau_n; \epsilon) \\
\text{PROJ} & \quad \Gamma \vdash M : \Pi (\tau_1; \ldots; \tau_i; R) \\
& \vdash \text{proj}_i M : \tau_i
\end{align*}
\]

These rules make sense with or without row variables.

Projection does not care about the fields beyond \(i\). Thanks to row variables, this can be expressed in terms of *parametric polymorphism*:

\[
\text{proj}_i : \forall \alpha. 1 \ldots \alpha_i \rho. \Pi (\alpha_1; \ldots; \alpha_i; \rho) \rightarrow \alpha_i
\]
About Rows

Rows were invented by Wand and improved by Rémy in order to ascribe precise types to operations on records.

The case of tuples, presented here, is simpler.

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

Rows are explained in depth by Pottier and Rémy [Pottier and Rémy, 2005].
Closure-passing closure conversion

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

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \exists \rho. \mu \alpha. \Pi \left( \alpha \times \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket ; \rho \right)$$

$\rho$ describes the environment
$\alpha$ is the concrete type of the closure
a tuple...
...that begins with a code pointer...
...and continues with the environment

See Morrisett and Harper’s “fix-type” encoding [1998].

**Question:** Why is it $\exists \rho. \mu \alpha. \tau$ and not $\mu \alpha. \exists \rho. \tau$

The type of the environment is fixed once for all and does not change at each recursive call.

**Question:** Notice that $\rho$ appears only once. Any comments?
Closure-passing closure conversion

Let $\text{Clo}(R)$ abbreviate $\mu\alpha.\Pi((\alpha \times [\tau_1]) \to [\tau_2]; R)$.

Let $U\text{Clo}(R)$ abbreviate its unfolded version, 
$\Pi((\text{Clo}(R) \times [\tau_1]) \to [\tau_2]; R)$.

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

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

\[
[M_1 M_2] = \text{let } \rho, \text{clo} = \text{unpack } [M_1] \text{ in} \\
\text{let code : (Clo(\rho) \times [\tau_1]) \to [\tau_2]} = \\
\text{proj}_0 (\text{unfold clo) in} \\
\text{code (clo, [M_2])}
\]
Closure-passing closure conversion

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

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

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

No extra field or extra work is required to store or construct a representation of the free variable \( f \): the closure itself plays this role.

However, this untyped code can only be typechecked when recursion is monomorphic.

**Exercise:**

Check well-typedness with monomorphic recursion.
Closure-passing closure conversion

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

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

Fortunately, we may slightly change the encoding, using a recursive closure as in the type-passing version, to allow typechecking in System F.
Closure-passing closure conversion

Let \( \tau \) be \( \forall \vec{\alpha}. \tau_1 \rightarrow \tau_2 \) and \( \Gamma_f \) be \( f : \tau, \Gamma \) where \( \vec{\beta} \neq \Gamma \)

\[
\llbracket \mu f : \tau. \lambda x. M \rrbracket = \text{let code = } \Lambda \vec{\beta}. \lambda (\text{clo : Clo}[\Gamma_f], x : [\tau_1]). \\
\text{let (\_code, f, x_1, \ldots, x_n) : } \forall \vec{\beta}. \text{UClo}(\llbracket \Gamma_f \rrbracket) = \\
\text{unfold clo in } \\
\llbracket M \rrbracket \text{ in } \\
\text{let rec clo : } \forall \vec{\beta}. \exists \rho. \text{Clo}(\rho) = \Lambda \vec{\beta}. \\
\text{pack } [\Gamma], (\text{fold (code } \vec{\beta}, \text{clo, x_1, } \ldots, x_n)) \text{ as } \exists \rho. \text{Clo}(\rho) \text{ in clo}
\]

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

Here, a polymorphic recursive function is directly compiled into a polymorphic recursive closure. Notice that the type of closures is unchanged so the encoding of applications is also unchanged.
Can we compile mutually recursive functions?

\[ M \overset{\triangle}{=} \mu(f_1, f_2). (\lambda x_1. M_1, \lambda x_2. M_2) \]

Environment passing:

\[ \llbracket M \rrbracket = \text{let } code_i = \lambda (env, x). \]
\[ \text{let } (f_1, f_2, x_1, \ldots, x_n) = env \text{ in } \]
\[ \llbracket M_i \rrbracket \]
\[ \text{in } \]
\[ \text{let rec } clo_1 = (code_1, (clo_1, clo_2, x_1, \ldots, x_n)) \]
\[ \text{and } clo_2 = (code_2, (clo_1, clo_2, x_1, \ldots, x_n)) \text{ in } \]
\[ clo_1, clo_2 \]
Mutually recursive functions

Can we compile mutually recursive functions?

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

Environment passing:

\[
\llbracket M \rrbracket = \begin{array}{l}
\text{let } code_i = \lambda (env, x). \\
\text{let } (f_1, f_2, x_1, \ldots, x_n) = env \text{ in } \\
\llbracket M_i \rrbracket \\
in \\
\text{let rec env } = (clo_1, clo_2, x_1, \ldots, x_n) \\
\text{and clo}_1 = (code_1, env) \\
\text{and clo}_2 = (code_2, env) \text{ in } \\
clo_1, clo_2
\end{array}
\]
Mutually recursive functions

Can we compile mutually recursive functions?

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

Environment passing:

\[
\begin{align*}
\text{let } code_i &= \lambda (\text{clo}, x). \\
&\quad \text{let } (\_ , f_1, f_2, x_1, \ldots, x_n) = \text{clo in } [M_i] \\
&\quad \text{in } \\
&\quad \text{let rec } clo_1 = (code_1, clo_1, clo_2, x_1, \ldots, x_n) \\
&\quad \quad \text{and } clo_2 = (code_2, clo_1, clo_2, x_1, \ldots, x_n) \\
&\quad \quad \text{in } clo_1, clo_2
\end{align*}
\]

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

Can we compile mutually recursive functions?

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

Environment passing:

\[
\begin{align*}
\text{let } code_1 &= \lambda (clo, x). \\
& \quad \text{let } (_\text{code}_1, _\text{code}_2, f_1, f_2, x_1, \ldots, x_n) = clo \text{ in } [M_1] \text{ in} \\
\text{let } code_2 &= \lambda (clo, x). \\
& \quad \text{let } (_\text{code}_2, f_1, f_2, x_1, \ldots, x_n) = clo \text{ in } [M_2] \text{ in} \\
\text{let rec } clo_1 &= (code_1, code_2, clo_1, clo_2, x_1, \ldots, x_n) \text{ and } clo_2 = clo_1.tail \\
\text{in } clo_1, clo_2
\end{align*}
\]

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

Can closure passing and environment passing be mixed?

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

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

\[
\begin{align*}
\llbracket \lambda x. M \rrbracket &= \text{let } code = \lambda(clo, x). \text{let } (_, (x_1, \ldots, x_n)) = clo \text{ in } \llbracket M \rrbracket \text{ in } (\text{code}, (x_1, \ldots, x_n)) \\
\llbracket M_1 M_2 \rrbracket &= \text{let } clo = \llbracket M_1 \rrbracket \text{ in } \text{let } code = \text{proj}_0 \text{ clo } \text{ in } \text{code} (clo, \llbracket M_2 \rrbracket)
\end{align*}
\]

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

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

A class definition is an object generator:

\[
\text{class } c \ (x_1, \ldots, x_q) \{
\text{meth } m_1 = M_1 \\
\quad \ldots \\
\text{meth } m_p = M_p \\
\}\]

Given arguments for parameter \( x_1, \ldots, x_1 \), it will build recursive methods \( m_1, \ldots, m_n \).
Encoding of objects

A class can be compiled into an object closure:

\[
\text{let } m = \\
\quad \text{let } m_1 = \lambda(m, x_1, \ldots, x_q). M_1 \text{ in} \\
\quad \ldots \\
\quad \text{let } m_p = \lambda(m, x_1, \ldots, x_q). M_p \text{ in} \\
\quad \{m_1, \ldots, m_p\} \text{ in} \\
\quad \lambda x_1 \ldots x_q. (m, x_1, \ldots x_q)
\]

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

Calling method \( m_i \) of an object \( p \) is

\[
(\text{proj}_0 p). m_i p
\]

How can we type the encoding?
Typed encoding of objects

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

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

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

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

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

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

An object has a recursive type but it is \textit{not} a recursive value.
Typed encoding of objects

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

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

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

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

Moral of the story

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

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

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

Because we have focused on type preservation, we have studied only na"ıve closure conversion algorithms.

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

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

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

- Algebraic Data Types
  - Equi- and iso-recursive types
- Typed closure conversion
- Existential types
  - Implicitly-type existential types passing
  - Iso-existential types
- Typed closure conversion
  - Environment passing
  - Closure passing
- Generalized Algebraic Datatypes
An introduction to GADTs
### Examples

```ocaml
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
```

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

```ocaml
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 y   -> let x = arg in apply Fadd (x, y)
  | Fbody     -> let b = arg in
    let step = Fstep (if not b then 1 else 2)
in apply step (apply step 0)
in apply Fbody true
```

---

### Defunctionalization

Introduce a constructor per call site

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

---

### GADTs

- Algebraic Data Types
- Typed closure conversion
- Existential types
- Typed closure conversion
- GADTs
Example

A typed abstract syntax tree

```ocaml
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)

```ocaml
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

```haskell
  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?
<table>
<thead>
<tr>
<th>Algebraic Data Types</th>
<th>Typed closure conversion</th>
<th>Existential types</th>
<th>Typed closure conversion</th>
<th>GADTs</th>
</tr>
</thead>
</table>

**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:

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

For instance, generic programming can be redefined as follows:

```ml
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!

```ml
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

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

- $M$ is typechecked with the assumption 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 assumption.
- 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 $F$

Coercions witness type equivalences:

variable

reflexivity

symmetry

transitivity

arrow coercions

left projection

right projection

type generalization

type instantiation

Types

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

Expressions

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

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

Typing rules

**Coerce**

$\Gamma \vdash M : \tau_1$  
$\Gamma \vdash \gamma : \tau_1 \sim \tau_2$  

$\Gamma \vdash \gamma \triangleleft M : \tau_2$

**Coercion**

$\Gamma \vdash \gamma : \tau_1 \sim \tau_2$

$\Gamma \vdash \gamma : \tau_1 \sim \tau_2$

**Coabs**

$\Gamma, x : \tau_1 \sim \tau_2 \vdash M : \tau$

$\Gamma \vdash \lambda x : \tau_1 \sim \tau_2. M : \tau$
Fc (simplified)

Conversion

\[
\text{EQ-HYP} \quad \frac{y : \tau_1 \sim \tau_2 \in \Gamma}{\Gamma \vdash y : \tau_1 \sim \tau_2}
\]

\[
\text{EQ-REF} \quad \frac{\Gamma \vdash \tau}{\Gamma \vdash \langle \tau \rangle : \tau \sim \tau}
\]

\[
\text{EQ-SYM} \quad \frac{\Gamma \vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash \text{sym} \gamma : \tau_2 \sim \tau_1}
\]

\[
\text{EQ-TRANS} \quad \frac{\Gamma \vdash \gamma_1 : \tau_1 \sim \tau \quad \Gamma \vdash \gamma_2 : \tau \sim \tau_2}{\Gamma \vdash \gamma_1; \gamma_2 : \tau_1 \sim \tau_2}
\]

\[
\text{EQ-LEFT} \quad \frac{\Gamma \vdash \gamma : \tau_1 \rightarrow \tau_2 \sim \tau_1' \rightarrow \tau_2'}{\Gamma \vdash \text{left} \gamma : \tau_1' \sim \tau_1}
\]

\[
\text{EQ-RIGHT} \quad \frac{\Gamma \vdash \gamma : \tau_1 \rightarrow \tau_2 \sim \tau_1' \rightarrow \tau_2'}{\Gamma \vdash \text{right} \gamma : \tau_2 \sim \tau_2'}
\]

\[
\text{EQ-ALL} \quad \frac{\Gamma, \alpha \vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash \forall \alpha. \gamma : \forall \alpha. \tau_1 \sim \forall \alpha. \tau_2}
\]

\[
\text{EQ-INST} \quad \frac{\Gamma \vdash \gamma : \forall \alpha. \tau_1 \sim \forall \alpha. \tau_2}{\Gamma \vdash \gamma@\tau : [\alpha \mapsto \tau] \tau_1 \sim [\alpha \mapsto \tau] \tau_2}
\]
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:

\[
(\gamma \triangleleft V_1) \ V_2 \quad \rightarrow \quad \text{right } \gamma \triangleleft (V_1 \ (\text{left } \gamma \triangleleft V_2))
\]
\[
(\gamma \triangleleft V) \ \tau \quad \rightarrow \quad (\gamma \triangleleft \tau) \triangleleft (V \ \tau)
\]
\[
\gamma_1 \triangleleft (\gamma_2 \triangleleft V) \quad \rightarrow \quad (\gamma_1; \gamma_2) \triangleleft V
\]
Coercions should be without computational content

Except for coercion abstractions that must stop the evaluation

- Otherwise, one could attempt to reduce \( M \) in \( \lambda \text{int} \sim \text{bool}. M \) when \( M \) is not (\text{bool} \triangleright 0), which is well-typed in this context.

- In call-by-value,
  \[
  \lambda x : \tau_1 \sim \tau_2. M \quad \text{freezes} \quad \text{the evaluation of} \ M,
  \\
  M \triangleright \gamma \quad \text{resumes} \quad \text{the evaluation of} \ M.
  \]

Must always be enforced, even with other strategies

- Full reduction \textit{at compile time} may still be performed, but be aware of stuck programs and treat them as dead branches.
Type soundness

By subject reduction and progress with explicit coercions

Erasing semantics

Important and non obvious.

\[ \gamma \ll M \text{ erases to } M \]
\[ \gamma \text{ erases to } \diamond \]

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
\[ \begin{align*}
\Gamma \vdash M : \tau_1 \\
\Gamma \vdash \tau_1 \sim \tau_2
\end{align*} \]
\[ \Gamma \vdash M : \tau_2 \]

Coercion
\[ \begin{align*}
\Gamma \vdash \tau_1 \sim \tau_2
\end{align*} \]
\[ \Gamma \vdash \diamond : \tau_1 \sim \tau_2 \]

Coabs
\[ \begin{align*}
\Gamma, x : \tau_1 \sim \tau_2 \vdash M : \tau
\end{align*} \]
\[ \Gamma \vdash \lambda x : \tau_1 \sim \tau_2. M : \tau \]
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

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 annotation 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 outside 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.


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.


