MPRI 2.4

Algebraic data types, existential types, and GADTs

François Pottier

2023

Towards data types

Many data types can be built out of sums and products and a form of recursion at the level of types.

Binary sum $+$ and product $\times$, and their neutral elements 0 and 1, suffice.

- The unit type is 1.
- The empty type is 0.
- The Boolean type is $1 + 1$.
- The type $\mathbb{N}$ of the natural numbers must satisfy $\mathbb{N} \simeq 1 + \mathbb{N}$.
- The type $\mathbb{L}(X)$ of lists of elements of type X must satisfy

$$\mathbb{L}(X) \simeq 1 + X \times \mathbb{L}(X)$$

Three technical approaches to data types

There are three main approaches to extending System *F* with data types:

- consider 0, 1, $+$, $\times$, and recursive types $\mu X.T$ as primitive concepts and encode all data types in terms of these concepts;
- consider algebraic data types as primitive and view sums, products, naturals, lists, etc., as instances of this general concept;
- introduce no new primitive concept and remark that inductive types can be encoded in System *F*.

In practice, the second approach is the most natural and user-friendly.

All three approaches, and their connections, are worth understanding.

Binary products

It is easy to add pairs and projections to the (call-by-value) λ-calculus.

$$\begin{array}{rcll} t & ::= & \ldots \mid (t,t) \mid \pi_i\ t & \quad\text{where } i \in \{1,2\} \\ v & ::= & \ldots \mid (v,v) & \\ E & ::= & \ldots \mid (E,t) \mid (v,E) \mid \pi_i\ E & \end{array}$$

One new reduction rule is needed: $\pi_i\ (v_1, v_2) \longrightarrow v_i$.

A new type constructor is needed: $T ::= \ldots \mid T \times T$.

Two new typing rules are needed:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2} \qquad\qquad \frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash \pi_i\ t : T_i}$$

Exercise: extend the proofs of Subject Reduction and Progress.

Unit

The unit type 1 can be viewed as a product type of arity 0.

It has an introduction form but no elimination form.

$$
\begin{array}{lcl}
t & ::= & \ldots \mid () \\
v & ::= & \ldots \mid ()
\end{array}
$$

– no new evaluation context

No new reduction rule is needed.

A new type constructor is needed: $T ::= \ldots \mid 1$.

One new typing rule is needed:

$$\Gamma \vdash () : 1$$

Binary sums

Let us add injections and a case analysis to (call-by-value) λ-calculus.

$$\begin{array}{rcll} t & ::= & \ldots \mid inj_i\ t \mid case\ t\ of\ t_1 \parallel t_2 & \quad \text{where } i \in \{1,2\} \\ v & ::= & \ldots \mid inj_i\ v & \\ E & ::= & \ldots \mid inj_i\ E \mid case\ E\ of\ t_1 \parallel t_2 & \end{array}$$

One new reduction rule is needed: $case\ inj_i\ v\ of\ t_1 \parallel t_2 \longrightarrow t_i\ v$.

In a *case* construct, the branches t_1 and t_2 should be functions.

A new type constructor is needed: $T ::= \ldots \mid T + T$.

Two new typing rules are needed:

$$\frac{\Gamma \vdash t : T_i}{\Gamma \vdash inj_i\ t : T_1 + T_2} \qquad \frac{\Gamma \vdash t : T_1 + T_2 \quad \Gamma \vdash t_1 : T_1 \to T' \quad \Gamma \vdash t_2 : T_2 \to T'}{\Gamma \vdash case\ t\ of\ t_1 \parallel t_2 : T'}$$

Exercise: extend the proofs of Subject Reduction and Progress.

Void

The empty type can be viewed as a sum type of arity 0.

It has an elimination form but no introduction form.

$$\begin{array}{rcl} t & ::= & \ldots \mid \mathit{absurd}\ t \\ & & \text{– no new value} \\ E & ::= & \ldots \mid \mathit{absurd}\ E \end{array}$$

No new reduction rule is needed. *absurd v* is stuck.

A new type constructor is needed: $T ::= \ldots \mid 0$.

One new typing rule is needed:

$$\frac{\Gamma \vdash t : 0}{\Gamma \vdash \mathit{absurd}\ t : T'}$$

Exercise: extend the proof of Progress.

Approaches to recursive types

Recall what was said earlier about recursive types:

- Natural numbers must satisfy $\mathbb{N} \simeq 1 + \mathbb{N}$.
- Lists must satisfy $\mathbb{L}(X) \simeq 1 + X \times \mathbb{L}(X)$.

One approach is to extend the type system with recursive types $\mu X.T$. The type $\mu X.T$ and its unfolding $T[\mu X.T/X]$ must then be considered either equal or related via explicit coercions.

One can then define $\mathbb{N}$ as $\mu X.\ 1 + X$ and $\mathbb{L}(X)$ as $\mu Y.\ 1 + X \times Y$.

A more pleasant approach is to just view $\mathbb{N}$ and $\mathbb{L}(X)$ as primitive types. This is the topic of the next slides, and leads to algebraic data types.

$\mathbb{N}$ as a primitive type

Consider λ-calculus with injections and case analysis.

Let us use $inj_1\ ()$ to encode zero and $inj_2\ v$ to encode the successor of v.

Introduce a new type constructor: $T ::= \ldots \mid \mathbb{N}$.

Give three new typing rules:

$$\frac{\Gamma \vdash t : 1}{\Gamma \vdash inj_1\ t : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash inj_2\ t : \mathbb{N}} \qquad \frac{\begin{array}{c}\Gamma \vdash t : \mathbb{N} \\ \Gamma \vdash t_1 : 1 \rightarrow T' \qquad \Gamma \vdash t_2 : \mathbb{N} \rightarrow T'\end{array}}{\Gamma \vdash case\ t\ of\ t_1 \parallel t_2 : T'}$$

These are exactly the typing rules proposed earlier for binary sums where we have replaced $T_1 + T_2$ with $\mathbb{N}$, T_1 with 1, and T_2 with $\mathbb{N}$.

The types $\mathbb{N}$ and $1 + \mathbb{N}$ are not equal, but they are isomorphic: one can write $in : 1 + \mathbb{N} \rightarrow \mathbb{N}$ and $out : \mathbb{N} \rightarrow 1 + \mathbb{N}$ such that $in \cdot out$ and $out \cdot in$ are $\beta\eta$-equal to the identity.

$\mathbb{L}(X)$ as a primitive type

Consider again λ-calculus with injections and case analysis.

Let us use $inj_1\ ()$ to encode [] and $inj_2\ (v_1, v_2)$ to encode $v_1 :: v_2$.

Introduce a new type constructor: $T ::= \ldots \mid \mathbb{L}(T)$.

Give three new typing rules:

$$\frac{\Gamma \vdash t : 1}{\Gamma \vdash inj_1\ t : \mathbb{L}(T)} \qquad \frac{\Gamma \vdash t : T \times \mathbb{L}(T)}{\Gamma \vdash inj_2\ t : \mathbb{L}(T)}$$

$$\frac{\Gamma \vdash t : \mathbb{L}(T) \qquad \Gamma \vdash t_1 : 1 \rightarrow T' \qquad \Gamma \vdash t_2 : T \times \mathbb{L}(T) \rightarrow T'}{\Gamma \vdash case\ t\ of\ t_1 \parallel t_2 : T'}$$

These are again exactly the typing rules of binary sums where we have replaced $T_1 + T_2$ with $\mathbb{L}(X)$, T_1 with 1, and T_2 with $X \times \mathbb{L}(X)$.

Again, the types $\mathbb{L}(X)$ and $1 + X \times \mathbb{L}(X)$ are isomorphic.

Algebraic data types

Instead of offering a fixed set of primitive types such as $\mathbb{N}$ and $\mathbb{L}(X)$, better let the user define whatever custom types they need using sums and products (of arbitrary arity) and recursion.

This idea gives rise to algebraic data types.

```
type      nat = Zero | Succ of nat
type 'a list = Nil  | Cons of 'a * 'a list
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

Named types, named data constructors, and pattern matching make algebraic data types extremely pleasant and safe to use.

Burstall, MacQueen, Sannella,
HOPE: An experimental applicative language, 1980.

Products and sums as algebraic data types

Sums and products can be viewed as algebraic data types.

```
type ('a, 'b) sum  = Left of 'a | Right of 'b
type void = |                           (* zero constructors *)
type ('a, 'b) pair = Pair of 'a * 'b
type unit = Unit
```

Deconstructing the type `void` works as expected:

```
let absurd (type a) (x : void) : a =
  match x with _ -> .                      (* zero branches *)
```

Encoding Booleans

The Boolean type $\mathbb{B} \simeq 1 + 1$ can be declared as an algebraic data type:

```
type bool = False | True
```

However, Booleans can also be encoded in pure λ-calculus.

A Boolean value is an "object with a *case* method".
It can choose between two branches:

$$
\begin{array}{rcl}
\mathbb{B} & \triangleq & \forall X.\, (1 \to X) \to (1 \to X) \to X \\
\mathit{False} & \triangleq & \lambda x_1.\, \lambda x_2.\, x_1\ () \\
\mathit{True} & \triangleq & \lambda x_1.\, \lambda x_2.\, x_2\ () \\
\mathit{case}\ t\ \mathit{of}\ t_1 \parallel t_2 & \triangleq & t\ t_1\ t_2
\end{array}
$$

This is a Scott encoding, and also a Church encoding.

Exercise: reconstruct the omitted type abstractions and applications.

Encoding sums

More generally, the binary sum type $T_1 + T_2$ can be encoded as follows:

$$
\begin{array}{rcl}
T_1 + T_2 & \triangleq & \forall X.\,(T_1 \to X) \to (T_2 \to X) \to X \\
\mathit{inj}_1\ x & \triangleq & \lambda x_1.\lambda x_2.\,x_1\ x \\
\mathit{inj}_2\ x & \triangleq & \lambda x_1.\lambda x_2.\,x_2\ x \\
\mathit{case}\ t\ \mathit{of}\ t_1 \parallel t_2 & \triangleq & t\ t_1\ t_2
\end{array}
$$

The zero-ary sum type 0 can be encoded, too!

$$
\begin{array}{rcl}
0 & \triangleq & \forall X.\,X \\
\mathit{absurd}\ t & \triangleq & t
\end{array}
$$

Clearly this works for any number of branches.

Encoding products

The binary product type $T_1 \times T_2$ can be encoded as follows:

$$
\begin{array}{rcl}
T_1 \times T_2 & \triangleq & \forall X.\,(T_1 \rightarrow T_2 \rightarrow X) \rightarrow X \\
(x_1, x_2) & \triangleq & \lambda k.\,k\ x_1\ x_2 \\
\pi_1\ t & \triangleq & t\ (\lambda x_1.\,\lambda x_2.\,x_1) \\
\pi_2\ t & \triangleq & t\ (\lambda x_1.\,\lambda x_2.\,x_2)
\end{array}
$$

The zero-ary product type 1 can be encoded, too!

$$
\begin{array}{rcl}
1 & \triangleq & \forall X.\,X \rightarrow X \\
() & \triangleq & \lambda x.x
\end{array}
$$

Clearly this works for any number of tuple components.

Encoding natural integers

Can we encode the recursive type $\mathbb{N} \simeq 1 + \mathbb{N}$ in the same way, à la Scott?

$$\mathbb{N} \quad \triangleq \quad \forall X.\,(1 \to X) \to (\mathbb{N} \to X) \to X$$

This doesn't work in System F, which doesn't have recursive types.

Here, the Scott and Church encodings differ.

The Church encoding views a number as "an object with a *fold* method".

$$\begin{array}{rcl} \mathbb{N} & \triangleq & \forall X.\,X \to (X \to X) \to X \\ \mathit{Zero} & \triangleq & \lambda z.\lambda s.\,z \\ \mathit{Succ}\;x & \triangleq & \lambda z.\lambda s.\,s\;(x\;z\;s) \end{array}$$

Encoding lists

The Church encoding views a list as “an object with a *fold* method”.

$$
\begin{array}{rcl}
\mathbb{L}(Y) & \triangleq & \forall X.\, X \to (Y \to X \to X) \to X \\
[] & \triangleq & \lambda n.\, \lambda c.\, n \\
x :: xs & \triangleq & \lambda n.\, \lambda c.\, c\ x\ (xs\ n\ c)
\end{array}
$$

The Church encoding works for all inductive types.

Girard, Taylor, Lafont, Proofs and types, 1990, §11.3–11.5.

Motivation

Complex numbers are an abstract concept.

Outside of their implementation, how they are represented should be irrelevant, and one should not depend on implementation details.

> *In one section, Professor Descartes announced that a complex number was an ordered pair of reals [...].*
>
> *In the other section, Professor Bessel announced that a complex number was an ordered pair of reals, the first of which was nonnegative [...].*
>
> *An unfortunate mistake [...] caused the two sections to be interchanged.*

Reynolds, Types, Abstraction and Parametric Polymorphism, 1983.

Complex numbers as an abstract type

In OCaml, one might implement complex numbers as an abstract type:

```
module Complex : sig
  type t
  val zero: t
  val one: t
  val add: t -> t -> t
  val mul: t -> t -> t
  val (=): t -> t -> bool
  (* etc. *)
end
```

Complex numbers as an existential type

In System F, this idea can be made precise via an existential type:

$$Complex : \exists X. \left\{ \begin{array}{r@{\,:\,}l} zero & X \\ add & X \to X \to X \\ mul & X \to X \to X \\ eq & X \to X \to bool \\ \multicolumn{2}{c}{etc.} \end{array} \right\}$$

Mitchell and Plotkin, Abstract types have existential type, 1988.

Rossberg, Russo, Dreyer, F-ing Modules, 2014.

Streams as an existential type

Imagine we wish to define an abstract type of streams.

A stream is a producer of a sequence of elements,
out of which a consumer can pull elements on demand.

It is an "object" with a single method, *next*.

- a stream has a certain current internal state.
- *next* returns either nothing or a pair of an element and a new state.

A stream is analogous to a Java iterator, except it is not mutable.
Its current state is explicit.

$$\mathbb{S}(X) \triangleq \exists S. (\underbrace{S \to 1 + X \times S}_{\textit{next}}) \times \underbrace{S}_{\textit{cur}}$$

Streams as an existential type

`('a, 's) step` corresponds to $1 + X \times S$:

```
type ('a, 's) step =
  | Done                          (* the stream is exhausted *)
  | Yield of 'a * 's    (* here is an element and a new state *)
```

OCaml views existential types as a special case of algebraic data types:

```
type 'a stream =
  | Stream:
            (* The [next] method: *) ('s -> ('a, 's) step) *
            (* The current state: *)  's
        (* together form a stream: *) -> 'a stream
```

The data constructor `Stream` has universal type: it is polymorphic in `'s`.

The producer chooses the type of the internal state;
the consumer must treat this type as abstract.

Converting a list to a stream

This conversion function is a nonrecursive producer:

```
let stream (xs : 'a list) : 'a stream =
  let next xs =
    match xs with
    |       [] -> Done
    | x :: xs -> Yield (x, xs)
  in
  Stream (next, xs)              (* packing an existential type *)
```

On the last line, what is the concrete type of states?

It is `'a list`.

Converting a stream to a list

This conversion function is a recursive consumer:

```
let unstream (Stream (next, s) : 'a stream) : 'a list =
  let rec unfold s =
    match next s with
    | Done          -> []
    | Yield (x, s) -> x :: unfold s
  in
  unfold s
```

The first line uses pattern matching to unpack an existential type.

What is the type of `unfold`?

It is `s -> 'a list`
where `s` is an abstract type introduced by unpacking at line 1.

Examples of stream producers

How would you implement a singleton stream?

```
let return (x : 'a) : 'a stream =
  let next s =
    if s then Yield (x, false) else Done
  in
  Stream (next, true)          (* packing an existential type *)
```

On the last line, the concrete type of states is `bool`:
either we have already yielded an element, or we have not.

Exercise: Write `interval` of type `int -> int -> int stream`.

Exercise: Write `append` of type `'a stream -> 'a stream -> 'a stream`.

An example consumer-and-producer

The `map` function on streams is also non-recursive:

```
let map (f : 'a -> 'b) (xs : 'a stream) : 'b stream =
  let Stream (next, s) = xs in                    (* unpacking *)
  let next s =
    match next s with
    | Done          -> Done
    | Yield (x, s) -> Yield (f x, s)
  in
  Stream (next, s)                                  (* packing *)
```

Streams as an existential type

This encoding of streams is used in practice.

In addition to `Done` and `Yield`, a third constructor `Skip` can be used, meaning "please ask again"

A consumer must ask, ask, ask until a non-`Skip` result is produced.

This allows most stream producers to be nonrecursive functions.
This makes optimization easier.

Coutts, Leshchinskiy, Stewart, Stream fusion: from lists to streams to nothing at all, 2007.

System F with existential types

The syntax of types is extended with existential types:

$$T ::= \ldots \mid \exists X.\, T$$

The syntax of terms is extended with introduction and elimination forms:

$$\begin{array}{rl} t ::= \ldots \mid \mathit{pack}\ T, t\ \ \mathit{as}\ \exists X.T \mid \mathit{let}\ X, x = \mathit{unpack}\ t\ \mathit{in}\ t \\ v ::= \ldots \mid \mathit{pack}\ T, v\ \ \mathit{as}\ \exists X.T \\ E ::= \ldots \mid \mathit{pack}\ T, E\ \ \mathit{as}\ \exists X.T \mid \mathit{let}\ X, x = \mathit{unpack}\ E\ \mathit{in}\ t \end{array}$$

A new reduction rule is introduced:

$$\begin{array}{rl} \mathit{let}\ X, x = \mathit{unpack}\ (\mathit{pack}\ T', v\ \mathit{as}\ \exists X.T)\ \mathit{in}\ t & \longrightarrow \\ t[v/x][T'/X] & \end{array}$$

System F with existential types

Two new typing rules are introduced:

∃-INTRO

$$\frac{\Gamma \vdash t : T[T'/X]}{\Gamma \vdash \mathit{pack}\ T', t\ \mathit{as}\ \exists X.T : \exists X.T}$$

∃-ELIM

$$\frac{\Gamma \vdash t_1 : \exists X.T \qquad X \# \Gamma, T_2 \qquad \Gamma; X; x : T \vdash t_2 : T_2}{\Gamma \vdash \mathit{let}\ X, x = \mathit{unpack}\ t_1\ \mathit{in}\ t_2 : T_2}$$

For reference, recall the typing rules for universal types:

∀-INTRO

$$\frac{\Gamma; X \vdash t : T \qquad X \# \Gamma}{\Gamma \vdash \Lambda X.t : \forall X.T}$$

∀-ELIM

$$\frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t\ T' : T[T'/X]}$$

Exercise: extend the proofs of Subject Reduction and Progress.

Universal/existential duality

When a value has universal type $\forall X.T$,
the producer of this value must treat X as abstract
and the consumer can choose a type T' with which to instantiate X.

When a value has existential type $\exists X.T$,
the producer chooses a type T' with which to instantiate X
but the consumer must treat X as abstract.

When a value has existential type, its consumer must be polymorphic.

Church encoding of existential types

Existential types can in fact be encoded in terms of universal types:

$$
\begin{array}{rcl}
\exists X.T & \triangleq & \forall Y.\,(\forall X.\,T \to Y) \to Y \\
\mathit{pack}\ T', v\ \mathit{as}\ \exists X.T & \triangleq & \Lambda Y.\,\lambda k : (\forall X.\,T \to Y).\,k\ T'\ v \\
\mathit{let}\ X, x = \mathit{unpack}\ t_1\ \mathit{in}\ t_2 : T_2 & \triangleq & t_1\ T_2\ (\Lambda X.\,\lambda x : T \to T_2.\,t_2)
\end{array}
$$

This encoding validates the logical implication $\exists X.T \to \neg\forall X.\neg T$ where $\neg T$ is defined as $T \to 0$.

Exercise: check that this encoding validates the reduction rule and the typing rules proposed earlier for primitive existential types.

Untyped expressions

Consider a tiny language of expressions $t ::= k \mid (t, t) \mid \pi_i\ t$:

```
type expr =
| EInt  of int
| EPair of expr * expr
| EFst  of expr
| ESnd  of expr
```

Expressions include integer constants, pairs, and projections.

Untyped values

A straightforward interpreter for this language uses a type of all values:

```
type value =
| VInt  of int
| VPair of value * value
```

This is an algebraic data type. Thus every value carries a tag.

Runtime tests

These tags are used in runtime tests that can cause runtime errors.

```
let as_pair (v : value) : value * value =
  match v with
  | VPair (v1, v2) ->
      v1, v2
  | _ ->
      assert false (* runtime error! *)
```

An untyped interpreter

Here, interpreting a pair projection operation involves a runtime test.

```
let rec eval (e : expr) : value =
  match e with
  | EInt x ->
      VInt x
  | EPair (e1, e2) ->
      VPair (eval e1, eval e2)
  | EFst e ->
      fst (as_pair (eval e))
  | ESnd e ->
      snd (as_pair (eval e))
```

This is necessary because this interpreter accepts untyped expressions.

Typed expressions

Let us impose a simple type discipline on expressions.

```
type _ expr =
| EInt  :                    int ->       int expr
| EPair : 'a expr * 'b expr -> ('a * 'b) expr
| EFst  :     ('a * 'b) expr ->          'a expr
| ESnd  :     ('a * 'b) expr ->          'b expr
```

This type definition encodes the following type discipline:

$$\Gamma \vdash k : int \qquad \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2} \qquad \frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash \pi_i\ t : T_i}$$

A meta-level AST of type `'a expr`
represents an object-level expression of type `'a`.

Typed values

Let us similarly impose a type discipline on values:

```
type _ value =
| VInt  :                          int ->        int value
| VPair : 'a value * 'b value -> ('a * 'b) value
```

Values are still tagged (for now), but runtime tests become unnecessary...

Look Ma, no runtime test!

Only one branch is now necessary. A second branch would be dead.

```
let as_pair : type a b . (a * b) value -> a value * b value
= function
  | VPair (v1, v2) ->
      v1, v2
  (* In this branch, we would learn [a * b = int], *)
  (* which is contradictory. *)
  (* | _ -> . *)
```

In OCaml, destructing a GADT requires a type annotation in this style.

A typed interpreter

Evaluating an expression of type T yields a value of type T.

```
let rec eval : type a . a expr -> a value
= function
  | EInt x ->
      (* We learn [a = int] so returning [VInt _] is OK. *)
      VInt x
  | EPair (e1, e2) ->
      (* For some types [a1] and [a2], we learn [a = a1 * a2] *)
      (* and we can assume [e1 : a1 expr] and [e2 : a2 expr]. *)
      VPair (eval e1, eval e2)
  | EFst e ->
      fst (as_pair (eval e))
  | ESnd e ->
      snd (as_pair (eval e))
```

The type of the interpreter reflects the subject reduction property.
… it amounts to checking the proof of subject reduction!

A typed, tagless interpreter

Evaluating an expression of type T yields a meta-level value of type T.

```
let rec eval : type a . a expr -> a
= function
  | EInt x ->
     (* We learn [a = int] so returning an integer is OK. *)
     x                   (* no tagging! *)
  | EPair (e1, e2) ->
     (* For some types [a1] and [a2], we learn [a = a1 * a2] *)
     (* and we can assume [e1 : a1 expr] and [e2 : a2 expr]. *)
     (eval e1, eval e2) (* no tagging! *)
  | EFst e ->
     fst (eval e)        (* no untagging! *)
  | ESnd e ->
     snd (eval e)        (* no untagging! *)
```

The type of the interpreter reflects the subject reduction property.
…g it amounts to checking the proof of subject reduction!

Going further

Our tiny expressions are closed: the typing judgement is $\vdash t : T$.

When expressions involve variables, one needs a type `('g, 'a) expr` whose definition encodes the typing judgement $\Gamma \vdash t : T$.

This is reasonably easy if variables are encoded as de Bruijn indices.

Bird, Paterson, de Bruijn notation as a nested datatype, 1999.

Runtime type descriptions

A value of type `'a ty` is a runtime description of the type `'a`.

```
type 'a ty =
| TyInt  :                            int ty
| TySum  : 'a ty * 'b ty -> ('a, 'b) sum ty
| TyPair : 'a ty * 'b ty ->    ('a * 'b) ty
```

Applications

Although inspecting a type at runtime is impossible, as types are erased, inspecting a runtime description of a type is possible.

In other words, although the type $\forall X.\, X \rightarrow X$ has only one inhabitant, the type $\forall X.\, \mathit{Ty}\; X \rightarrow X \rightarrow X$ has more than one.

This let us write polymorphic, type-directed functions, an activity that is sometimes known as generic programming.

Show

Here is a polymorphic, type-directed conversion of a value to a string.

```
let rec show : type a . a ty -> a -> string =
  fun ty x ->
    match ty with
    | TyInt ->
        string_of_int x
    | TySum (ty1, ty2) ->
        begin match x with
        | Left x1  -> "left("  ^ show ty1 x1 ^ ")"
        | Right x2 -> "right(" ^ show ty2 x2 ^ ")"
        end
    | TyPair (ty1, ty2) ->
        let (x1, x2) = x in
        "(" ^ show ty1 x1 ^ ", " ^ show ty2 x2 ^ ")"
```

In each branch, we learn something about the type of x.

Show

It is more concise and looks better to deconstruct both arguments at once.

```
let rec show : type a . a ty -> a -> string =
  fun ty x ->
    match ty, x with
    | TyInt, x ->
        string_of_int x
    | TySum (ty1, _), Left x1 ->
        "left("  ^ show ty1 x1 ^ ")"
    | TySum (_, ty2), Right x2 ->
        "right(" ^ show ty2 x2 ^ ")"
    | TyPair (ty1, ty2), (x1, x2) ->
        "(" ^ show ty1 x1 ^ ", " ^ show ty2 x2 ^ ")"
```

The OCaml type-checker reads patterns from left to right
so deconstructing `(ty, x)` works but deconstructing `(x, ty)` does not.

Show

Here is a polymorphic, type-directed equality test.

```
let rec equal : type a . a ty -> a -> a -> bool =
  fun ty x y ->
    match ty, x, y with
    | TyInt, x, y ->
        Int.equal x y
    | TySum (ty1, _), Left x1, Left y1 ->
        equal ty1 x1 y1
    | TySum (_, ty2), Right x2, Right y2 ->
        equal ty2 x2 y2
    | TySum _, Left _, Right _
    | TySum _, Right _, Left _ ->
        false
    | TyPair (ty1, ty2), (x1, x2), (y1, y2) ->
        equal ty1 x1 y1 && equal ty2 x2 y2
let rec equal : type a b . a ty -> b ty -> (a, b) eq =
  2 ->
```

57/115

Connections between GADTs and type classes

`Eq` and `Show` are typical examples of type classes in Haskell.

Upcoming lecture on type classes (PED).

Hinze, Jeuring, Löh,
Comparing Approaches to Generic Programming in Haskell, 2006.

Connections between GADTs and type classes

A closed set of type class instances can be compiled down to GADTs.

Pottier and Gauthier,
Polymorphic typed defunctionalization and concretization, 2006.

However a GADT describes a closed universe of structural types whereas type classes are open-ended and apply to user-defined, nominal types.

The Holy Grail is to propose a language where a type of the representations of all types (including itself!) can be defined.

Chapman, Dagand, McBride, Morris,
The Gentle Art of Levitation, 2010.

A type inferencer

We have a type `'a expr` of well-typed expressions
and a type `'a ty` of runtime type descriptions.

Can we express a simple type type inferencer that accepts an untyped expression and either fails or returns a typed expression?

```
exception IllTyped
let rec infer : Raw.expr -> ???
= function
  | Raw.EInt i ->
      (TyInt, EInt i)
  | ...
```

What should its result type be?

A type inferencer

We need an existential type $\exists X.\ \mathit{Ty}\ X \times \mathit{Expr}\ X$.

```
type typed_expr =
| Pack   : 'a ty * 'a expr -> typed_expr
```

A type inferencer

We can now write the type inferencer:

```
let rec infer : Raw.expr -> typed_expr =
  function
  | Raw.EInt i ->
      Pack (TyInt, EInt i)
  | Raw.EFst e ->
      let Pack (ty, e) = infer e in
      begin match ty with
      | TyPair (ty1, ty2) -> Pack (ty1, EFst e)
      | _                 -> raise IllTyped
      end
```

Exercise: write the two missing cases.

A type-checker

Can we check whether an expression has a certain expected type?

We would like to write something like this:

```
let check (type a) (e : Raw.expr) (expected : a ty) : a expr =
  let Pack (inferred, e) = infer e in
  if inferred = expected then
    e
  else
    raise IllTyped
```

But this code is not well-typed. Why?

`expected` has type `a ty`.

`inferred` has type `b ty`
where `b` is an unknown type introduced by deconstructing `Pack`.

They cannot be compared using homogoneous equality `=`.

Even if they could, `e` has type `b expr`
whereas a result of type `a expr` is required.

The equality GADT

The solution involves the type equality GADT.

```
type (_, _) eq =
| Equal: ('a, 'a) eq
```

The type `('a, 'b) eq` has at most one inhabitant.

If it has one then this inhabitant must be `Equal`
and the types `'a` and `'b` must be the same.

A heterogenous type equality test

This lets us express a heterogenous type equality test:

```
let rec equal : type a b . a ty -> b ty -> (a, b) eq =
  fun ty1 ty2 ->
    match ty1, ty2 with
    | TyInt, TyInt ->
        Equal
    | TyPair (ty1a, ty1b), TyPair (ty2a, ty2b) ->
        let Equal = equal ty1a ty2a in
        let Equal = equal ty1b ty2b in
        Equal
    | _, _ ->
        raise IllTyped
```

When `equal ty1 ty2` succeeds, we learn that the runtime type descriptions `ty1` and `ty2` describe the same static type.

Exercise: write the missing case.

A type-checker

We can now write the type-checker:

```
let check (type a) (e : Raw.expr) (expected : a ty) : a expr =
  let Pack (inferred, e) = infer e in
  let Equal = equal inferred expected in
  e
```

Exercise: make sure that you understand why this code is well-typed.

Putting the pieces together

Given an arbitrary untyped expression in our tiny language, we can now infer its type, evaluate it, and show its value, whatever its type may be.

```
let () =
  let e = Raw.(EPair (EInt 42, EInt 0)) in
  let Pack (ty, e) = infer e in
  let v = eval e in
  Printf.printf "%s\n%!" (show ty v)
```

The output in the REPL is:

```
(42, 0)
```

Printf in OCaml

`printf` takes a "format string" followed with a number of arguments:

```
# open Printf;;
# printf "%d * %s = %d\n" 2 "12" 24;;
2 * 12 = 24
- : unit = ()
```

The number and type of these arguments depends on the format string.

Printf in OCaml

A format string is actually not a string: it is a data structure.

```
# open CamlinternalFormatBasics;;
# let desc : _ format6 = "%d * %s = %d\n";;
val desc :
  (int -> string -> int -> 'a, 'b, 'c, 'd, 'd, 'a) format6 =
  Format
   (Int (Int_d, No_padding, No_precision,
     String_literal (" * ",
      String (No_padding,
       String_literal (" = ",
        Int (Int_d, No_padding, No_precision,
         Char_literal ('\n', End_of_format)))))),
   "%d * %s = %d\n")
```

Printf in OCaml

This data structure has the shape of a list:

```
Int (Int_d, No_padding, No_precision,
String_literal (" * ",
String (No_padding,
String_literal (" = ",
Int (Int_d, No_padding, No_precision,
Char_literal ('\n',
End_of_format))))))
```

`End_of_format` is "nil"; the other constructors are "cons" constructors.

`Int` and `String` correspond to "holes" `%d` and `%s`.

`String_literal` and `Char_literal` correspond to literal pieces of string.

An algebraic data type of descriptors

Can we define our own algebraic data type of formats, or descriptors?

```
type desc =
  | Nil
  | Lit of string * desc
  | Int of desc
```

An algebraic data type of descriptors

Or, in this alternative syntax:

```
type desc =
  | Nil   :                        desc
  | Lit   : string * desc -> desc
  | Int   : desc          -> desc
```

An algebraic data type of descriptors

Or, in this alternative syntax:

```
type desc =
  | Nil   :                      desc
  | Lit   : string * desc -> desc
  | Int   : desc          -> desc
```

Now, please define `fprintf` so that `fprintf emit desc <args>`

- emits output via the function `emit : string -> unit`,
- obeys `desc`,
- expects arguments `<args>` whose number and type satisfy `desc`.

`fprintf` should have type `(string -> unit) -> desc -> ??? -> unit`.

Expressing the type of fprintf

The type `desc -> ??? -> unit` does not make sense.

The number of and type of the arguments `???` depends on the descriptor.

We seem to need a dependent type `(d: desc) -> shape d`

- where `shape` would be a function of descriptors to types,
- but OCaml does not have that.

Instead, let's use a plain function type `'shape desc -> 'shape`

- where the definition of `'shape desc` as a GADT encodes the correspondence between descriptors and shapes.

Descriptors form a typed language and `fprintf` is an interpreter for it!

A GADT of descriptors

We want `fprintf : (string -> unit) -> 'a desc -> 'a`.

```
type   desc =
  | Nil  :                                    desc
  | Lit  :            string *    desc ->     desc
  | Int  :                        desc ->     desc
```

We must turn the type `desc` into a GADT.

A GADT of descriptors

We want `fprintf : (string -> unit) -> 'a desc -> 'a`.

```
type _ desc =
  | Nil  :                                       ?? desc
  | Lit  :         string * ?? desc ->           ?? desc
  | Int  :                  ?? desc ->           ?? desc
```

We parameterize the type `desc`.

A GADT of descriptors

We want `fprintf : (string -> unit) -> 'a desc -> 'a`.

```
type _ desc =
  | Nil  :                                  unit desc
  | Lit  :          string * ?? desc ->       ?? desc
  | Int  :                   ?? desc ->       ?? desc
```

`Nil` requires no action; the corresponding shape is `unit`.

A GADT of descriptors

We want `fprintf : (string -> unit) -> 'a desc -> 'a`.

```
type _ desc =
  | Nil  :                                 unit desc
  | Lit  :          string * 'a desc ->      'a desc
  | Int  :                   ?? desc ->      ?? desc
```

`Lit (s, d)` requires printing `s` and interpreting `d`.

If `d` has shape `'a` then `Lit (s, d)` has shape `'a` as well.

A GADT of descriptors

We want `fprintf : (string -> unit) -> 'a desc -> 'a`.

```
type _ desc =
  | Nil   :                                       unit desc
  | Lit   :          string * 'a desc ->            'a desc
  | Int   :                   'a desc ->   (int -> 'a) desc
```

`Int d` requires consuming an integer argument and interpreting `d`.

If `d` has shape `'a` then `Int d` has shape `int -> 'a`.

A GADT of descriptors

We want `fprintf : (string -> unit) -> 'a desc -> 'a`.

```
type _ desc =
  | Nil  :                                                unit desc
  | Lit  :                  string * 'a desc ->             'a desc
  | Hole : ('data -> string) * 'a desc -> ('data -> 'a) desc
```

We change the hole of type `int` with a hole of arbitrary type `'data`.

All that is needed is a conversion function of type `'data -> string`.

Implementing fprintf

```
let fprintf (type a) emit (desc : a desc) : a =
  let rec eval : type a . a desc -> a =
    function
    | Nil ->
        ???
    | Lit (s, desc) ->
        ???
    | Hole (to_string, desc) ->
        ???

  in eval desc
```

Recall

```
  | Nil  :                                          unit desc
```

Implementing fprintf

```
let fprintf (type a) emit (desc : a desc) : a =
  let rec eval : type a . a desc -> a =
    function
    | Nil ->
        ??? (* We learn [a = unit]. *)
    | Lit (s, desc) ->
        ???
    | Hole (to_string, desc) ->
        ???

  in eval desc
```

Recall

```
  | Nil  :                                    unit desc
```

Implementing fprintf

```
let fprintf (type a) emit (desc : a desc) : a =
  let rec eval : type a . a desc -> a =
    function
    | Nil ->
        ()  (* We learn [a = unit]. *)
    | Lit (s, desc) ->
        ???
    | Hole (to_string, desc) ->
        ???

  in eval desc
```

Recall

```
  | Lit   :               string * 'a desc ->            'a desc
```

Implementing fprintf

```
let fprintf (type a) emit (desc : a desc) : a =
  let rec eval : type a . a desc -> a =
    function
    | Nil ->
        ()
    | Lit (s, desc) ->
        ??? (* We learn no new type equality. *)
    | Hole (to_string, desc) ->
        ???

  in eval desc
```

Recall

```
  | Lit   :             string * 'a desc ->             'a desc
```

Implementing fprintf

```
let fprintf (type a) emit (desc : a desc) : a =
  let rec eval : type a . a desc -> a =
    function
    | Nil ->
        ()
    | Lit (s, desc) ->
        emit s; eval desc
    | Hole (to_string, desc) ->
        ???

  in eval desc
```

Recall

```
  | Hole : ('data -> string) * 'a desc -> ('data -> 'a) desc
```

Implementing fprintf

```
let fprintf (type a) emit (desc : a desc) : a =
  let rec eval : type a . a desc -> a =
    function
    | Nil ->
        ()
    | Lit (s, desc) ->
        emit s; eval desc
    | Hole (to_string, desc) ->
        ??? (* We learn [a = data -> b] *)
            (* [to_string : data -> string; desc : b desc] *)
  in eval desc
```

Recall

```
  | Hole : ('data -> string) * 'b desc -> ('data -> 'b) desc
```

Implementing fprintf

```
let fprintf (type a) emit (desc : a desc) : a =
  let rec eval : type a . a desc -> a =
    function
    | Nil ->
        ()
    | Lit (s, desc) ->
        emit s; eval desc
    | Hole (to_string, desc) ->
        fun x -> emit (to_string x); eval desc
        (* [x] has type [data]; [eval desc] has type [b] *)
  in eval desc                      (* and [data -> b] is [a] *)
```

Recall

```
  | Hole : ('data -> string) * 'b desc -> ('data -> 'b) desc
```

Using `fprintf`

Voilà! From `fprintf`, we get `printf`.

```
let printf desc =
  let emit = print_string in
  fprintf emit desc
```

Using `fprintf`

To construct descriptors, some sugar is needed.

```
module Sugar = struct
  let nil = Nil
  let lit s desc = Lit (s, desc)
  let d desc = Hole (string_of_int, desc)
  let s desc = Hole (Fun.id, desc)
end
```

Using `fprintf`

To construct descriptors, some sugar is needed.

```
module Sugar = struct
  let nil = Nil
  let lit s desc = Lit (s, desc)
  let d desc = Hole (string_of_int, desc)       (* %d *)
  let s desc = Hole (Fun.id, desc)              (* %s *)
end
```

For example,

```
let desc =                                    (* "%d * %s = %d\n" *)
  d @@ lit " * " @@ s @@ lit " = " @@ d @@ lit "\n" @@ nil
```

Using `fprintf`

```
let desc =                                   (* "%d * %s = %d\n" *)
  d @@ lit " * " @@ s @@ lit " = " @@ d @@ lit "\n" @@ nil
```

Try this in the OCaml REPL (read-eval-print-loop):

```
# let () = printf desc 2 "12" 24;;
2 * 12 = 24
```

Implementing sprintf

Can we implement `sprintf`, which returns a string?

```
let sprintf desc args =
  let b = Buffer.create 128 in
  let emit = Buffer.add_string b in
  fprintf emit desc args;
  Buffer.contents b
```

This is accepted but is not what we want.

Implementing sprintf

Can we implement `sprintf`, which returns a string?

```
let sprintf desc <arg ... arg> =
  let b = Buffer.create 128 in
  let emit = Buffer.add_string b in
  fprintf emit desc <arg ... arg>;
  Buffer.contents b
```

We want `sprintf` to accept a variable number of arguments, not just one.

In fact, we cannot write the type of `sprintf`.

It is like the type of `fprintf` but should end in **`string`** instead of **`unit`**.

A more general type of descriptors

We must equip ourselves with a more general type of descriptors.

```
type _ desc =
  | Nil  :                                               unit desc
  | Lit  :                  string *  'a       desc ->       'a desc
  | Hole : ('data -> string) *  'a       desc ->
                                        ('data -> 'a     ) desc
```

A more general type of descriptors

We must equip ourselves with a more general type of descriptors.

```
type (_, _) desc =
  | Nil  :                                          ('r, 'r) desc
  | Lit  :                string * ('a, 'r) desc -> ('a, 'r) desc
  | Hole : ('data -> string) * ('a, 'r) desc ->
                                       ('data -> 'a, 'r) desc
```

In the type `('a, 'r) desc`,

- `'a` is the shape, as before,
- `'r` is the eventual return type of this shape.
  - it can be `unit` for `fprintf` and `string` for `sprintf`;
  - a descriptor can be polymorphic in `'r`.

Implementing fprintf, again

We can now give `fprintf` a more general type. We parameterize it with:

- `emit : string -> unit`
- `finished : unit -> r` — new
- `desc : (a, r) desc`

`fprintf emit finished desc` has type `a`.

`a` must in fact be a function type whose eventual return type is `r`.

`fprintf emit finished desc <args>` must eventually return a value of type `r`, which it obtains by calling `finished()`.

Implementing fprintf, again

```
let fprintf (type a r) emit (finished : unit -> r)
                         (desc : (a, r) desc) : a =
  let rec eval : type a . (a, r) desc -> a =
    function
    | Nil ->
        (* We have [a = r] so [finished()] has type [a]. *)
        finished()
    | Lit (s, desc) ->
        emit s; eval desc
    | Hole (to_string, desc) ->
        fun x -> emit (to_string x); eval desc
  in eval desc
```

It is worth pointing out that `eval` involves polymorphic recursion.

Implementing printf and sprintf

We can now implement `printf` and `sprintf`, among other variations:

```
let printf desc =
  let emit = print_string
  and finished () = () in
  fprintf emit finished desc

let sprintf desc =
  let b = Buffer.create 128 in
  let emit = Buffer.add_string b
  and finished () = Buffer.contents b in
  fprintf emit finished desc
```

We get

```
val printf : ('a, unit) desc -> 'a
val sprintf : ('a, string) desc -> 'a
```

Using `printf` and `sprintf`

```
let desc () =                            (* "%d * %s = %d\n" *)
  d @@ lit " * " @@ s @@ lit " = " @@ d @@ lit "\n" @@ nil
```

Try this in the OCaml REPL (read-eval-print-loop):

```
# let () = printf (desc()) 2 "12" 24;;
2 * 12 = 24
# let (s : string) = sprintf (desc()) 2 "12" 24;;
val s : string = "2 * 12 = 24\n"
```

Here, we make `desc` a (constant) function in order to work around the value restriction. See upcoming lecture on mutable state (GS).

Danvy et al.'s approach

Danvy, Keller and Puech (2015) view formats as trees instead of lists.

```
type (_, _) desc =
  | Lit  :                           string ->            ('a, 'a) desc
  | Hole :          ('data -> string) -> ('data -> 'a, 'a) desc
  | Seq  : ('a, 'b) desc * ('b, 'c) desc ->       ('a, 'c) desc
```

The type `('a, 'r) desc` has the same meaning as earlier.

`Lit` and `Hole` no longer play the role of list "cons" constructors.

`Seq` is a binary concatenation constructor, whose type says:

If `'a` *is a multi-arrow type whose eventual return type is* `'b` *and*
if `'b` *is a multi-arrow type whose eventual return type is* `'c` *then*
`'a` *is a multi-arrow type whose eventual return type is* `'c`*.*

Danvy et al.'s approach

Danvy et al. write `kprintf` in continuation-passing style:

```
let rec kprintf
: type a r . (a, r) desc -> (string -> r) -> a =
  fun desc finished ->
    match desc with
    | Lit s ->
        finished s
    | Hole to_string ->
        fun x -> finished (to_string x)
    | Seq (desc1, desc2) ->
        kprintf desc1 @@ fun s1 ->
        kprintf desc2 @@ fun s2 ->
        finished (s1 ^ s2)
```

Exercise (easy): define `printf`, `sprintf`, and `fprintf` using `kprintf`.

Exercise (harder): define `fprintf` directly.
Do not use string concatenation `^`.

System F+GADTs

System F+GADTs was defined by Xi, Chen and Chen (2003).

Xi, Chen, Chen,
Guarded Recursive Datatype Constructors, 2003.

Pottier and Gauthier,
Polymorphic typed defunctionalization and concretization, 2006.

System F+GADTs: the typing judgement

Recall the typing judgement of System F:

$$\Gamma \vdash t : T$$

In System F+GADTs, must we change the shape of this judgement?

We must extend it with a conjunction of equality hypotheses.

$$\Gamma, C \vdash t : T$$

Equality constraints are given by $C, D ::= \mathit{True} \mid T = T \mid C \wedge C$.

System F+GADTs: type declarations

We assume that a family of type constructors F is given.

- for simplicity, we assume they have arity 1.

We assume that a family of data constructors K is given.

- for simplicity, we assume they have arity 1.

We assume that each data constructor has a closed type scheme:

$$K \; : \; \forall \bar{X}.\, D \Rightarrow T_1 \rightarrow F\ T_2$$

System F+GADTs: an auxiliary judgement

For readability, we introduce the auxiliary judgement

$$K \;\leq\; D \Rightarrow T_1 \to F\ T_2$$

whose definition is the following:

$$\frac{K \;:\; \forall \bar{X}.\, D \Rightarrow T_1 \to F\ T_2}{K \;\leq\; D[\bar{T}/\bar{X}] \Rightarrow T_1[\bar{T}/\bar{X}] \to F\ T_2[\bar{T}/\bar{X}]}$$

System F+GADTs: the typing judgement

The typing rules of System F are unchanged. A constraint is transported.

$$\textsc{Var}\quad \Gamma, C \vdash x : \Gamma(x)$$

$$\textsc{Abs}\quad \frac{\Gamma; x : T_1, C \vdash t : T_2}{\Gamma, C \vdash \lambda x.t : T_1 \to T_2}$$

$$\textsc{App}\quad \frac{\Gamma, C \vdash t_1 : T_1 \to T_2 \qquad \Gamma, C \vdash t_2 : T_1}{\Gamma, C \vdash t_1\ t_2 : T_2}$$

$$\textsc{TAbs}\quad \frac{\Gamma; X, C \vdash t : T \qquad X \# \Gamma}{\Gamma, C \vdash \Lambda X.t : \forall X.T}$$

$$\textsc{TApp}\quad \frac{\Gamma, C \vdash t : \forall X.T}{\Gamma, C \vdash t\ T' : T[T'/X]}$$

System F+GADTs: the typing judgement, continued

The typing rule for a data constructor application is straightforward:

$$\textsc{DCon}\quad \frac{K \;\leq\; D \Rightarrow T_1 \rightarrow F\ T_2 \qquad C \Vdash D \qquad \Gamma, C \vdash t : T_1}{\Gamma, C \vdash K\ t : F\ T_2}$$

We write $C \Vdash D$ when C entails D (see next slide).

System F+GADTs: entailment

Let ρ denote a total mapping of type variables to closed types.

We write $\rho \vdash C$ when ρ satisfies C:

$$\rho \vdash \mathit{True} \qquad \frac{\rho(T_1) = \rho(T_2)}{\rho \vdash T_1 = T_2} \qquad \frac{\rho \vdash C_1 \qquad \rho \vdash C_2}{\rho \vdash C_1 \wedge C_2}$$

Entailment is then defined by:

$$\frac{\forall \rho.\ \rho \vdash C \Rightarrow \rho \vdash D}{C \Vdash D}$$

Entailment is decidable.

System F+GADTs: the typing judgement, continued

Type-checking a case analysis construct is straightforward:

$$\textsc{Case}\quad \frac{\begin{array}{c}\Gamma, C \vdash t : T_1 \\ \forall c \in \bar{c}.\quad \Gamma, C \vdash c : T_1 \rightarrow T_2 \\ \bar{c} \text{ is exhaustive}\end{array}}{\Gamma, C \vdash \mathit{case}\ t\ \mathit{of}\ \bar{c} : T_2}$$

A clause takes the form $c ::= K\ \bar{X}\ x \mapsto t$.

$\bar{c}$ is exhaustive if it contains a clause for every data contructor K.

System F+GADTs: the typing judgement, continued

When a clause is entered, new constraints appear locally.

$$\textsc{Clause}\quad \frac{\begin{array}{c} K : \forall\bar{X}.\, D \Rightarrow T_1 \rightarrow F\ T_2 \\ (\Gamma; \bar{X}; x : T_1), (C \wedge D \wedge F\ T_2 = F\ T_2') \vdash t : T' \\ \bar{X} \# \Gamma,\ C,\ T_2',\ T' \end{array}}{\Gamma, C \vdash K\ \bar{X}\ x \mapsto t : F\ T_2' \rightarrow T'}$$

System F+GADTs: the typing judgement, continued

There remains to introduce a typing rule that exploits the hypothesis C:

$$\textsc{Conversion}\quad \frac{\Gamma, C \vdash t : T \qquad C \Vdash T = T'}{\Gamma, C \vdash t : T'}$$

This rule is not syntax-directed.

One can imagine a variant of the system where conversion is explicit.
System FC is the core language of the Glasgow Haskell compiler.

Sulzmann, Chakravarty, Peyton Jones, Donnelly,
System F with Type Equality Coercions, 2007.

System F+GADTs: type soundness

Exercise: write down the omitted details (e.g., the reduction rule for *case*), then prove Subject Reduction and Progress.