

MPRI FUN  
Generalized  
Algebraic  
Data Types

François  
Pottier

Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

Metatheory

Conclusion

MPRI FUN

GADTs

François Pottier



2025–2026

## 1 Examples

A well-typed tagless interpreter

Runtime type descriptions

A well-typed type-checker

Type-checking printf and friends

## 2 Metatheory

## 3 Conclusion

## 1 Examples

A well-typed tagless interpreter

Runtime type descriptions

A well-typed type-checker

Type-checking printf and friends

## 2 Metatheory

## 3 Conclusion

## Untyped expressions

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

### Conclusion

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

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

### Conclusion

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:

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

A **meta-level** abstract syntax tree (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.  
Type-checking 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.  
Type-checking it amounts to checking the proof of subject reduction!

# Polymorphic recursion

## Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

## Metatheory

Conclusion

eval involves polymorphic recursion:  
the fact that eval is polymorphic  
is exploited in the definition of eval itself.

For example, when applied to an expression of type a Expr,  
eval calls itself recursively with an expression of type a1 Expr.

## Polymorphic recursion

Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

Metatheory

Conclusion

In System  $F$  with recursive functions (in Curry style), polymorphic recursion is just recursion. In this rule,  $T_1$  can be a universal type:

$$\frac{\text{LETREC} \quad \Gamma; f : T_1 \vdash \lambda x. t_1 : T_1 \quad \Gamma; f : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let rec } f = \lambda x. t_1 \text{ in } t_2 : T_2}$$

In Church style, one would allow  $\Lambda \bar{X}.$  to appear in front of  $\lambda x. t_1.$

**Exercise:** show that one could also view  $\text{let rec } f = \lambda x. t_1 \text{ in } t_2$  as sugar for  $\text{let } f = \text{fix } (\lambda f. \lambda x. t_1) \text{ in } t_2,$  where  $\text{fix}$  is a constant. What reduction rule and typing rule should be given for this constant?

## Polymorphic recursion

Examples

Tagless  
interpretersRuntime type  
descriptionsA well-typed  
type-checker

Printf &amp; friends

Metatheory

Conclusion

In ML (OCaml, Haskell, etc.), types  $T$  and type schemes  $S ::= \forall \bar{X}. T$  are distinguished. **FORALL-INTRO** is used at *let* constructs only.

$$\text{FORALL-INTRO} \quad \frac{\Gamma \vdash t : T \quad \bar{X} \# \Gamma}{\Gamma \vdash t : \forall \bar{X}. T}$$

$$\text{FORALL-ELIM} \quad \frac{\Gamma \vdash t : \forall \bar{X}. T}{\Gamma \vdash t : T[\bar{T}/\bar{X}]}$$

$$\text{LETRECMONO} \quad \frac{\Gamma; f : T_1 \vdash \lambda x. t_1 : T_1 \quad \bar{X} \# \Gamma \quad \Gamma; f : \forall \bar{X}. T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let rec } f = \lambda x. t_1 \text{ in } t_2 : T_2}$$

$$\text{LETRECPOLY} \quad \frac{\Gamma; f : S_1 \vdash \lambda x. t_1 : S_1 \quad \Gamma; f : S_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let rec } f : S_1 = \lambda x. t_1 \text{ in } t_2 : T_2}$$

Polymorphic recursion requires a type annotation.

Mycroft, [Polymorphic type schemes and recursive definitions](#), 1984.

## 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.

## 1 Examples

A well-typed tagless interpreter

Runtime type descriptions

A well-typed type-checker

Type-checking printf and friends

## 2 Metatheory

## 3 Conclusion

## Runtime type descriptions

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

### Conclusion

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

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

The binary sum type `('a, 'b) sum` is defined as follows:

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

It is also available in the standard library module **Either**.

# An example of a runtime type description

```
let example : (int * int) ty =
  TyPair (TyInt, TyInt)
```

The value `example` is a runtime description of the type `int * int`.

This value has no other type.

This type has no other inhabitant: `(int * int) ty` is a singleton type.

## 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  $\forall X. \text{Ty } X \rightarrow X \rightarrow X$  has more than one.

This lets 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.

## Equal

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
```

# Connections between GADTs and type classes

## Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

## Metatheory

## Conclusion

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

Here, a somewhat similar effect is achieved using GADTs.

Upcoming lecture on type classes (DR).

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

## Connections between GADTs and type classes

Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker  
Printf & friends

Metatheory

Conclusion

A **fixed** set of type class instances can be compiled down to a GADT.

If a Haskell program contains three instances of the class **Show**,  
for integers, products, and sums,  
then compiling type classes to GADTs  
would produce (roughly) the function `show` of the previous slide.

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

A fixed set of **functions** can also be compiled down to a GADT.

– See above paper and next week's lecture!

## Connections between GADTs and type classes

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

Conclusion

A limitation of this compilation scheme is that  
a GADT describes a **fixed** universe of types  
whereas type classes are **open-ended**.

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.

## 1 Examples

A well-typed tagless interpreter

Runtime type descriptions

A well-typed type-checker

Type-checking printf and friends

## 2 Metatheory

## 3 Conclusion

## Untyped expressions

We have a type of raw (untyped) expressions:

```
module Raw = struct
  type expr =
    | EInt of int
    | EPair of expr * expr
    | EFst of expr
    | ESnd of expr
end
```

This is an ordinary algebraic data type.

# Typed expressions and type descriptions

We also have a type `'a expr` of well-typed expressions:

```
type _ expr =
| EInt : int -> int expr
| ...
```

and a type `'a ty` of runtime type descriptions:

```
type _ ty =
| TyInt : int ty
| ...
```

## Question

Can we write a simple 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  $\exists X. \text{Ty } X \times \text{Expr } X$ .

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

Exercise: write the two missing cases (**EPair** and **ESnd**).

## 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 TypedExpr (inferred, e) = infer e in  
    if inferred = expected then  
      e  
    else  
      raise IllTyped
```

But **this code is not well-typed. Why?**

## 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 TypedExpr (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 **TypedExpr**.

## 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 TypedExpr (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 **TypedExpr**.

They **cannot be compared** using homogeneous 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.

For example, the type (**int** , **int**) eq has one inhabitant, namely **Equal**.  
The type (**int** , **bool**) eq has no inhabitant.

## The equality GADT

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

The data constructor **Equal** has polymorphic type:

$$\forall \alpha. (\alpha, \alpha) \text{ eq}$$

which can also be understood as a **constrained** polymorphic type:

$$\forall \alpha \beta. (\alpha = \beta) \Rightarrow (\alpha, \beta) \text{ eq}$$

Any color the customer wants, as long as it's black. – Henry Ford

# A heterogeneous type equality test

Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

Metatheory

Conclusion

This lets us express a [heterogeneous](#) 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 TypedExpr (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 TypedExpr (ty, e) = infer e in
  let v = eval e in
  Printf.printf "%s\n!" (show ty v)
```

This program prints:

(42, 0)

# Putting the pieces together

Here is  
a second example  
(of a different type!):

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

This program prints:

42

## 1 Examples

A well-typed tagless interpreter

Runtime type descriptions

A well-typed type-checker

Type-checking printf and friends

## 2 Metatheory

## 3 Conclusion

## 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

## Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

## Metatheory

## Conclusion

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

## Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

## Metatheory

## Conclusion

Or, in this alternative syntax:

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

# An algebraic data type of descriptors

## Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

## Metatheory

## Conclusion

Or, in this alternative syntax:

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

This is a little language of instructions.

**Nil** is the empty sequence of instructions.

**Lit** (s, d) prints the string s and continues with d.

**Int** d consumes an integer argument, prints it, and continues with d.

# An algebraic data type of descriptors

## Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

## Metatheory

## Conclusion

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.

## 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

## Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

## Metatheory

## Conclusion

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

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

### Conclusion

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

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

### Conclusion

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`.

We can in fact replace `Int` with a more general constructor `Hole...`

# 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 now allow a hole of arbitrary type `'data`.

We require 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 sprintf (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 sprintf (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 sprintf (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
```

Its type is 'a desc -> 'a.

## 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

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

### Conclusion

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

@@ is OCaml's low-priority **application operator**.

## 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 naive_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 naive_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.

Its (inferred) type is `('a -> 'b) desc -> 'a -> string`.

## Implementing sprintf

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

### Conclusion

Can we implement sprintf, which returns a string?

```
let naive_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.

Its (inferred) type is `('a -> 'b) desc -> 'a -> string`.

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

## Implementing sprintf

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

Conclusion

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

Whereas `fprintf` has type `desc -> ??? -> unit`  
which we have encoded as `'a desc -> 'a`,  
we want `sprintf` to have type `desc -> ??? -> string`.

How can this be expressed?

## A more general type of descriptors

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

### Conclusion

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

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

## A more general type of descriptors

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

### Conclusion

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 sprintf, again

```
let sprintf (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
```

## 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  $('a, 'r) \text{ desc}$  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  $\wedge$ .

## 1 Examples

A well-typed tagless interpreter

Runtime type descriptions

A well-typed type-checker

Type-checking printf and friends

## 2 Metatheory

## 3 Conclusion

## 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: terms

The syntax of terms is extended with constructor applications and case analysis constructs:

$$\begin{array}{lcl} t & ::= & \dots \\ & | & K t \\ & | & \text{case } t \text{ of } \bar{c} \\ c & ::= & K \bar{X} x \mapsto t \end{array}$$

Each branch in a case construct is a **clause**  $c$ .

# 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?

## 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.

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

This means: under  $\Gamma$ , assuming the constraint  $C$  is true,  $t$  has type  $T$ .

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

In a well-formed judgement, every variable or type variable in  $C, t, T$  is introduced by  $\Gamma$ .

## System $F$ +GADTs: type declarations

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

### Conclusion

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

- for simplicity, we assume they have arity 1.

The syntax of types includes applications of type constructors:

$$T := X \mid T \rightarrow T \mid \forall X. T \mid T + T \mid T \times T \mid 0 \mid 1 \mid F\ T$$

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](#):

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

## Example

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

Conclusion

For example, the equality GADT *eq* could be a type constructor:

$$F ::= \dots \mid eq$$

*Equal* could be a data constructor:

$$K ::= \dots \mid Equal$$

whose type scheme would be:

$$Equal : \forall \alpha \beta. \alpha = \beta \Rightarrow unit \rightarrow eq (\alpha \times \beta)$$

This is a constrained type scheme – remember Henry Ford.

## System F+GADTs: an auxiliary judgement

For readability, we introduce the auxiliary judgement

$$K \leq D \Rightarrow T_1 \rightarrow F T_2$$

whose definition is the following:

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

This judgement means that  $D \Rightarrow T_1 \rightarrow F T_2$  is a valid monomorphic constrained type for  $K$ .

For example, we have

$$\frac{\text{Equal} : \forall \alpha \beta. \alpha = \beta \Rightarrow \text{unit} \rightarrow \text{eq } (\alpha \times \beta)}{\text{Equal} \leq \text{int} = \text{int} \Rightarrow \text{unit} \rightarrow \text{eq } (\text{int} \times \text{int})}$$

## System $F$ +GADTs: the typing judgement

The shape of the typing judgement is now  $\Gamma \mid C \vdash t : T$ .

This means: under  $\Gamma$ , assuming the constraint  $C$  is true,  $t$  has type  $T$ .

$$\begin{array}{c}
 \text{VAR} \\
 \Gamma \mid C \vdash x : \Gamma(x)
 \end{array}
 \quad
 \begin{array}{c}
 \text{ABS} \\
 \frac{\Gamma; x : T_1 \mid C \vdash t : T_2}{\Gamma \mid C \vdash \lambda x. t : T_1 \rightarrow T_2}
 \end{array}
 \quad
 \begin{array}{c}
 \text{APP} \\
 \frac{\begin{array}{c} \Gamma \mid C \vdash t_1 : T_1 \rightarrow T_2 \\ \Gamma \mid C \vdash t_2 : T_1 \end{array}}{\Gamma \mid C \vdash t_1 \ t_2 : T_2}
 \end{array}$$

$$\begin{array}{c}
 \text{TABS} \\
 \frac{\Gamma; X \mid C \vdash t : T \quad X \not\in C}{\Gamma \mid C \vdash \Lambda X. t : \forall X. T}
 \end{array}
 \quad
 \begin{array}{c}
 \text{TAPP} \\
 \frac{\Gamma \mid C \vdash t : \forall X. T}{\Gamma \mid C \vdash t \ T' : T[T'/X]}
 \end{array}$$

The rules of System  $F$  are unchanged, except a constraint is transported.

System *F*+GADTs: the typing judgement,  
continued

Examples

Tagless  
interpretersRuntime type  
descriptionsA well-typed  
type-checker

Printf &amp; friends

Metatheory

Conclusion

$$\frac{\begin{array}{c} \text{DCOn} \\ K \leq D \Rightarrow T_1 \rightarrow F T_2 \\ C \Vdash D \\ \hline \Gamma \mid C \vdash t : T_1 \end{array}}{\Gamma \mid C \vdash K t : F T_2}$$

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

For example, we have

$$\frac{\begin{array}{c} Equal \leq int = int \Rightarrow unit \rightarrow eq (int \times int) \\ True \Vdash int = int \quad \Gamma \mid True \vdash () : unit \\ \hline \Gamma \mid True \vdash Equal () : eq (int \times int) \end{array}}{\Gamma \mid True \vdash Equal () : eq (int \times int)}$$

On the other hand,  $\Gamma \mid True \vdash Equal () : eq (int \times bool)$  cannot be proved because  $True \Vdash int = bool$  does not hold.

## System F+GADTs: entailment

Let  $\rho$  denote a total mapping of type variables to closed types.

We write  $\rho \vdash C$  to mean that  $\rho$  satisfies  $C$  or  $\rho$  is a solution of  $C$ :

$$\frac{\rho \vdash \text{True}}{\rho \vdash T_1 = T_2} \qquad \frac{\rho \vdash C_1 \quad \rho \vdash C_2}{\rho \vdash C_1 \wedge C_2}$$

The entailment  $C \Vdash D$  holds if every solution of  $C$  is also solution of  $D$ :

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

Entailment is decidable.

## System $F$ +GADTs: entailment

For example,

$\text{True}$	entails	$\text{int} = \text{int}$
$\alpha = \text{int}$	entails	$\text{int} = \alpha$
$\alpha = \text{int} \wedge \alpha = \beta$	entails	$\beta = \text{int}$
$\alpha \times \beta = \text{int} \times \text{bool}$	entails	$\alpha = \text{int} \wedge \beta = \text{bool}$
$\alpha \times \beta = \text{int} \times \text{bool}$	entails	$\alpha \times \alpha = \text{int} \times \text{int}$
$\text{int} = \text{bool}$	entails	$\text{False}$
$F T = F' T'$	entails	$\text{False}$ if $F \neq F'$
$\text{False}$	entails	$T = T'$

## System *F*+GADTs: the typing judgement, continued

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

Conclusion

Type-checking a case analysis construct is straightforward:

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

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

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

System  $F$ +GADTs: the typing judgement,  
continued

Examples

Tagless  
interpretersRuntime type  
descriptionsA well-typed  
type-checker

Printf &amp; friends

Metatheory

Conclusion

When a clause is entered, new constraints appear locally.

CLAUSE

$$\frac{\begin{array}{c} K : \forall \bar{X}. D \Rightarrow T_1 \rightarrow F T_2 \\ (\Gamma; \bar{X}; x : T_1) \mid (C \wedge D \wedge F T_2 = F' T'_2) \vdash t : T' \\ \bar{X} \# C, T'_2, T' \end{array}}{\Gamma \mid C \vdash K \bar{X} x \mapsto t : F' T'_2 \rightarrow T'}$$

For example,

$$\frac{\begin{array}{c} Equal : \forall \alpha \beta. \alpha = \beta \Rightarrow unit \rightarrow eq(\alpha \times \beta) \\ (\Gamma; \alpha; \beta; x : unit) \mid (True \wedge \alpha = \beta \wedge eq(\alpha \times \beta) = eq(T \times U)) \vdash t : T' \end{array}}{\Gamma \mid True \vdash Equal \alpha \beta x \mapsto t : eq(T \times U) \rightarrow T'}$$

so the branch  $t$  is type-checked under the assumption  
 $eq(\alpha \times \beta) = eq(T \times U)$ , which entails  $T = U$ .The type-checker can assume  $T = U$  while type-checking  $t$ .

## System *F*+GADTs: the typing judgement, continued

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

Conclusion

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

$$\frac{\text{CONVERSION} \quad \Gamma \mid C \vdash t : T \quad C \Vdash T = T'}{\Gamma \mid C \vdash t : T'}$$

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: the typing judgement, continued

### Examples

Tagless  
interpreters

Runtime type  
descriptions

A well-typed  
type-checker

Printf & friends

### Metatheory

Conclusion

$$\frac{\text{CONTRADICTION} \\ C \Vdash \text{False}}{\Gamma \mid C \vdash t : T}$$

Ex falso, quod libet.

In particular, if  $C \Vdash \text{False}$  then  $\Gamma \mid C \vdash \text{absurd} : T$ .

In OCaml, *absurd* is written . and is used to indicate a dead branch.

# System $F$ +GADTs: type soundness

## Examples

- Tagless interpreters
- Runtime type descriptions
- A well-typed type-checker
- Printf & friends

## Metatheory

Conclusion

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

## 1 Examples

A well-typed tagless interpreter

Runtime type descriptions

A well-typed type-checker

Type-checking printf and friends

## 2 Metatheory

## 3 Conclusion

## Worth remembering

Last week, we have seen that algebraic data types combine:

- sums, products, recursive types,
- and can also encode existential types.

In addition, generalized algebraic data types can encode

- type equality witnesses.

Adding just the type equality GADT  $(\alpha, \beta) \text{ eq}$  would suffice.

## Worth remembering

The fundamental new feature of GADTs is to let a **value** serve as a witness of an equality between two **types**.

A **runtime test** (a case analysis) can reveal type information, even though types do not exist at runtime: **type erasure** is still possible.

GADTs allow simulating some uses of **dependent types**.

Without GADTs, one could live with ordinary algebraic data types: more **tags**, more (redundant) **runtime tests**, more **dead branches**.

With GADTs, one can gain space, time, elegance and static assurance.