GADTs gone mild

Code at `https://gabriel.radanne.net/talks`

Gabriel Radanne

Definition: Generalized Algebraic Data Types (GADT)

The least maintainable way of writing interpreters[1]

[1] Except maybe dependent types

ADT: Algebraic Data Types

Types with sum and products:

```
type list =
  | Nil
  | Cons of int * list
```

Parametrized Algebraic Data Types

Parametrized types with sum and products:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

Parametrized Algebraic Data Types

Parametrized types with sum and products:

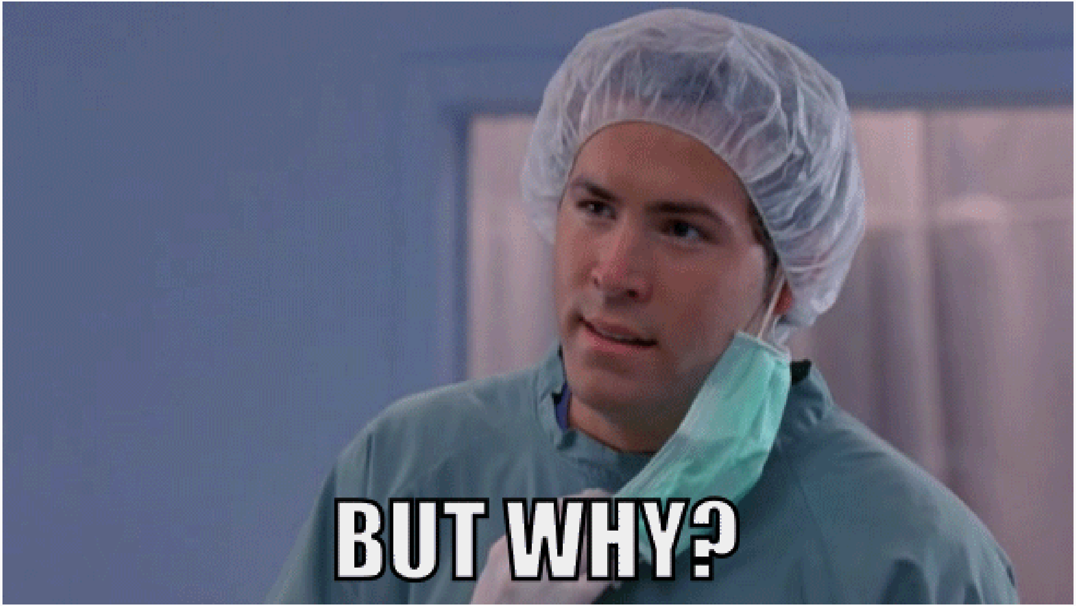
```
type 'a list =
  | Nil : 'a list
  | Cons : 'a * 'a list -> 'a list
```

Generalized Algebraic Data Types

Types with sum and products where we can change the return type:

```
type _ t =
  | A : string t
  | B : int -> float t

let x : float t = B 2
let y : string t = A
```


BUT WHY?

Compact arrays

Let's say we want to have compact arrays[2]:

```
type 'a t =
  | Array of 'a array
  | String of string (* This is more compact! *)
```

```
let get x i = match x with
  | Array a -> Array.get a i
  | String s -> String.get s i
```

You get the following type signature:

```
val get : char t -> int -> char
```

This is too specific!

[2] Example courtesy of Yaron Minsky "Why GADTs matter for performance"

Compact arrays

Let's say we want to have compact arrays[2]:

```
type 'a t =
  | Array of 'a array
  | String of string (* This is more compact! *)
let get x i = match x with
  | Array a -> Array.get a i
  | String s -> String.get s i
```

You get the following type signature:

```
val get : char t -> int -> char
```

This is too specific!

[2] Example courtesy of Yaron Minsky "Why GADTs matter for performance"

Compact arrays

Let's say we want to have compact arrays[2]:

```
type 'a t =
  | Array of 'a array
  | String of string (* This is more compact! *)
let get x i = match x with
  | Array a -> Array.get a i
  | String s -> String.get s i
```

You get the following type signature:

```
val get : char t -> int -> char
```

This is too specific!

[2] Example courtesy of Yaron Minsky "Why GADTs matter for performance"

Compact arrays – with GADTs

Let's say we want to have compact arrays:

```
type 'a t =
  | Array : 'a array -> 'a t
  | String : string -> char t
```

Compact arrays – with GADTs

Let's say we want to have compact arrays:

```
type 'a t =
  | Array : 'a array -> 'a t
  | String : string -> char t
```

```
let get
: type a. a t -> int -> a (* ∀α.α t → int → α *)
= fun x i -> match x with
  | Array a -> Array.get a i
  | String s -> String.get s i
val get : 'a t -> int -> 'a
```

The type annotation is necessary!

Compact arrays – with GADTs

```
# let x = String "Topinambour!" ;;
val x : char t
# get x 3 ;;
- : char = 'i'
# let y = Array [|1;2|] ;;
val y : int t
# get y 0 ;;
- : int = 1
```

Compact arrays – with GADTs

```
# let x = String "Topinambour!" ;;
val x : char t
# get x 3 ;;
- : char = 'i'
# let y = Array [|1;2|] ;;
val y : int t
# get y 0 ;;
- : int = 1
```

Do you want to build an interpreter?

Expressions

Let's write a small interpreter!

Our language will have:

- Boolean and integers constants
- If expressions
- Addition
- Equality test

Expressions – Type definition

```
type expr =
  | Int of int (* 42 *)
  | Bool of bool (* true *)
  | Add of expr*expr (* e + e *)
  | If of expr*expr*expr (* if b then e else e*)
  | Equal of expr*expr (* e = e *)

(* if 1 = 2 then 3 else 4 *)
If (Equal (Int 1, Int 2), Int 3, Int 4)
```

```
let rec eval e = match e with
  | Int i -> i
  | Bool b -> (* ... *)
```

```
let rec eval e = match e with
  | Int i -> i
  | Bool b -> (* ... *)
```

```
type value = I of int | B of bool

let rec eval e = match e with
  | Int i -> I i
  | Bool b -> B b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    (match v1, v2 with
     | I i1, I i2 -> I (i1 + i2)
     | _ -> failwith "Moule a gaufres!")
  | If (b, e1, e2) ->
    (match eval b with
     | B true -> eval e1
     | B false -> eval e2
     | I _ -> failwith "Anacoluthe!")
  | Equal _ -> (* ... *)
```

```
type value = I of int | B of bool

let rec eval e = match e with
  | Int i -> I i
  | Bool b -> B b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    (match v1, v2 with
     | I i1, I i2 -> I (i1 + i2)
     | _ -> failwith "Moule a gaufres!")
  | If (b, e1, e2) ->
    (match eval b with
     | B true -> eval e1
     | B false -> eval e2
     | I _ -> failwith "Anacoluthe!")
  | Equal _ -> (* ... *)
```

```ocaml
type value = I of int | B of bool

let rec eval e = match e with
  | Int i -> I i
  | Bool b -> B b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    (match v1, v2 with
    | I i1, I i2 -> I (i1 + i2)
    | _ -> failwith "Moule a gaufres!")
  | If (b, e1, e2) ->
    (match eval b with
    | B true -> eval e1
    | B false -> eval e2
    | I _ -> failwith "Anacoluthe!")
  | Equal _ -> (* ... *)
```

```
type value = I of int | B of bool

let rec eval e = match e with
  | Int i -> I i
  | Bool b -> B b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    (match v1, v2 with
     | I i1, I i2 -> I (i1 + i2)
     | _ -> failwith "Moule a gaufres!")
  | If (b, e1, e2) ->
    (match eval b with
     | B true -> eval e1
     | B false -> eval e2
     | I _ -> failwith "Anacoluthe!")
  | Equal _ -> (* ... *)
```

```ocaml
type value = I of int | B of bool

let rec eval e = match e with
  | Int i -> I i
  | Bool b -> B b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    (match v1, v2 with
    | I i1, I i2 -> I (i1 + i2)
    | _ -> failwith "Moule a gaufres!")
  | If (b, e1, e2) ->
    (match eval b with
    | B true -> eval e1
    | B false -> eval e2
    | I _ -> failwith "Anacoluthe!")
  | Equal _ -> (* ... *)
```

Expressions – problems

Problems:

- It's annoying to write
- It scales poorly to many different values
- The OCaml type system doesn't help us

Enter GADTs!

Expressions – problems

Problems:

- It's annoying to write
- It scales poorly to many different values
- The OCaml type system doesn't help us

Enter GADTs!

Expressions – the GADT way

We add a new type parameter

```
type 'a expr =
  | Int: int -> int expr
  | Bool: bool -> bool expr
  | Add: int expr * int expr -> int expr
  | If: bool expr * 'a expr * 'a expr -> 'a expr
  | Equal: 'a expr * 'a expr -> bool expr

(* if 1 = 2 then 3 else 4 *)
let e : int expr =
  If (Equal (Int 1, Int 2), Int 3, Int 4)
```

```
let rec eval
: type a. a expr -> a (* ∀α. α expr → α *)
= fun e -> match e with
  | Int i -> i
  | Bool b -> b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    v1 + v2
  | If (b, e1, e2) ->
    if eval b then eval e1 else eval e2
  | Equal (e1, e2) -> (eval e1 = eval e2)
# eval e ;;
- : int = 4
```

Tada!

```
let rec eval
: type a. a expr -> a (* ∀α. α expr → α *)
= fun e -> match e with
  | Int i -> i
  | Bool b -> b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    v1 + v2
  | If (b, e1, e2) ->
    if eval b then eval e1 else eval e2
  | Equal (e1, e2) -> (eval e1 = eval e2)
# eval e ;;
- : int = 4
```

Tada!

```
let rec eval
: type a. a expr -> a (* ∀α. α expr → α *)
= fun e -> match e with
  | Int i -> i
  | Bool b -> b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    v1 + v2
  | If (b, e1, e2) ->
    if eval b then eval e1 else eval e2
  | Equal (e1, e2) -> (eval e1 = eval e2)
# eval e ;;
- : int = 4
```

Tada!

```
let rec eval
: type a. a expr -> a (* ∀α. α expr → α *)
= fun e -> match e with
  | Int i -> i
  | Bool b -> b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    v1 + v2
  | If (b, e1, e2) ->
    if eval b then eval e1 else eval e2
  | Equal (e1, e2) -> (eval e1 = eval e2)
# eval e ;;
- : int = 4
```

Tada!

```
let rec eval
: type a. a expr -> a (* ∀α. α expr → α *)
= fun e -> match e with
  | Int i -> i
  | Bool b -> b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    v1 + v2
  | If (b, e1, e2) ->
    if eval b then eval e1 else eval e2
  | Equal (e1, e2) -> (eval e1 = eval e2)
# eval e ;;
- : int = 4
```

Tada!

```
let rec eval
: type a. a expr -> a (* ∀α. α expr → α *)
= fun e -> match e with
  | Int i -> i
  | Bool b -> b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    v1 + v2
  | If (b, e1, e2) ->
    if eval b then eval e1 else eval e2
  | Equal (e1, e2) -> (eval e1 = eval e2)
# eval e ;;
- : int = 4
```

Tada!

```
let rec eval
: type a. a expr -> a (* ∀α. α expr → α *)
= fun e -> match e with
  | Int i -> i
  | Bool b -> b
  | Add (e1,e2) ->
    let v1 = eval e1 and v2 = eval e2 in
    v1 + v2
  | If (b, e1, e2) ->
    if eval b then eval e1 else eval e2
  | Equal (e1, e2) -> (eval e1 = eval e2)
# eval e ;;
- : int = 4
```

Tada!

Expressions with GADTs

This is usually called HOAS (High Order Abstract Syntax).

Pros:

- It's *so cool*.
- The type system checks that your evaluation function is correct.
- Validity of expressions is encoded in the type system.

Cons:

- You can only express things that are valid in the host type system.
- Moving from the untyped world to the typed world is difficult.
  `parse : string -> ? expr`
- Transformations must be type preserving.
- It doesn't scale **at all** with the complexity of the domain.

⇒ Almost only usable for toy languages. Otherwise, it creates an unmaintainable mess.

Expressions with GADTs

This is usually called HOAS (High Order Abstract Syntax).

Pros:

- It's *so cool*.
- The type system checks that your evaluation function is correct.
- Validity of expressions is encoded in the type system.

Cons:

- You can only express things that are valid in the host type system.
- Moving from the untyped world to the typed world is difficult.
  ```
  parse : string -> ? expr
  ```
- Transformations must be type preserving.
- It doesn't scale **at all** with the complexity of the domain.

⇒ Almost only usable for toy languages. Otherwise, it creates an unmaintainable mess.

Expressions with GADTs

This is usually called HOAS (High Order Abstract Syntax).

Pros:

- It's *so cool*.
- The type system checks that your evaluation function is correct.
- Validity of expressions is encoded in the type system.

Cons:

- You can only express things that are valid in the host type system.
- Moving from the untyped world to the typed world is difficult.
  ```
  parse : string -> ? expr
  ```
- Transformations must be type preserving.
- It doesn't scale **at all** with the complexity of the domain.

$\Rightarrow$ Almost only usable for toy languages. Otherwise, it creates an unmaintainable mess.

Results on GADTs, aka. Poor man's dependent types

Invented by 3 different groups:

- Augustsson & Petersson (1994): Silly Type Families
- Cheney & Hinze (2003): First-Class Phantom Types.
- Xi, Chen & Chen (2003): Guarded Recursive Datatype Constructors.

Type *inference* is undecidable.

Checking of exhaustiveness in pattern matching is undecidable (Garrigue and Le Normand (2015): GADTs and Exhaustiveness: Looking for the Impossible).

Interaction with subtyping is a mess (Scherer, Rémy (2013) GADTs Meet Subtyping).

Type error messages become quite baroque.

Examples of use for GADTs

There is a large body of literature with examples of use for GADTs:

- How to program toy interpreters with GADTs in the most unreadable way
- How to encode unary numbers in types in the most verbose way
- Some far and few attempts at doing something actually useful (usually not in publications, amusingly)[3].

[3] This critique does not apply to the literature on dependent types.

But what can we actually do with GADTs?

Things you can encode in GADTs

- Existential types

  ```
  type t = Exists : 'a -> t (* ∃α. α *)
  ```

- Type level (Unary) Natural numbers
- Type level lists
- Type level finite sets
- Type level tree-like inclusion hierarchies
- Small Typed DSLs
- …
- Any property expressible by a context free language

Things you can encode in GADTs

- Existential types

  ```
  type t = Exists : 'a -> t (* ∃α. α *)
  ```

- Type level (Unary) Natural numbers
- Type level lists
- Type level finite sets
- Type level tree-like inclusion hierarchies
- Small Typed DSLs
- ...
- Any property expressible by a context free language

Things you can encode in GADTs

- Existential types

  ```
  type t = Exists : 'a -> t (* ∃α. α *)
  ```
- Type level (Unary) Natural numbers
- Type level lists
- Type level finite sets
- Type level tree-like inclusion hierarchies
- Small Typed DSLs
- ...
- Any property expressible by a context free language by encoding a pushdown automaton.

Things you can encode in GADTs

- Existential types

```
type t = Exists : 'a -> t (* ∃α. α *)
```

- Type level (Unary) Natural numbers
- Type level lists
- Type level finite sets
- Type level tree-like inclusion hierarchies
- Small Typed DSLs
- ...
- Any property expressible by a context free language by encoding a pushdown automaton.
    - And some contextual grammars ($a^n b^n c^n$)
    - Or worse (solutions to PCP)

Things you can encode in GADTs

- Existential types

  ```
  type t = Exists : 'a -> t (* ∃α. α *)
  ```

- Type level (Unary) Natural numbers
- Type level lists
- Type level finite sets
- Type level tree-like inclusion hierarchies
- Small Typed DSLs
- ...
- Any property expressible by a context free language by encoding a pushdown automaton.
  - And some contextual grammars ($a^n b^n c^n$)
  - Or worse (solutions to PCP)

Things you can encode in GADTs

- Existential types

```
type t = Exists : 'a -> t (* ∃α. α *)
```

- Type level (Unary) Natural numbers
- Type level lists
- Type level finite sets
- Type level tree-like inclusion hierarchies
- Small Typed DSLs
- ...
- Any property expressible by a context free language by encoding a pushdown automaton.
  - And some contextual grammars ($a^n b^n c^n$)
  - Or worse (solutions to PCP)

Printf

Printf – The best bad idea in the C standard library

```
printf(
  "We have %d potatoes which weight %f kg.",
  5, 1.2);
```

First argument is a string with holes

- `%d` is an integer hole
- `%f` is a floating point hole

Then, takes as many arguments as there are holes.

Printf – In OCaml

In OCaml, we also have printf:

```
Format.printf
  "We have %d potatoes which weight %f kg."
  5 1.2
```

This is *statically* checked.

[3] We use the `Format` module here. The `Printf` module is best avoided.

Where is the magic?

```
# printf ;;
- : ('a, formatter, unit) format -> 'a
# printf "%d sabords!" 1000;;
1000 sabords!
# printf "%d sabords!" 10.5;;
Error: This expression has type float but
an expression was expected of type int
# printf "%d sabords!";;
- : int -> unit
# fun s -> printf s 1000;;
- : (int -> 'a, formatter, unit) format -> 'a
```

Wat.

Where is the magic?

```
# printf ;;
- : ('a, formatter, unit) format -> 'a
# printf "%d sabords!" 1000;;
1000 sabords!
# printf "%d sabords!" 10.5;;
Error: This expression has type float but
an expression was expected of type int
# printf "%d sabords!";;
- : int -> unit
# fun s -> printf s 1000;;
- : (int -> 'a, formatter, unit) format -> 'a
```

Wat.

Where is the magic?

```
# printf ;;
- : ('a, formatter, unit) format -> 'a
# printf "%d sabords!" 1000;;
1000 sabords!
# printf "%d sabords!" 10.5;;
Error: This expression has type float but
an expression was expected of type int
# printf "%d sabords!";;
- : int -> unit
# fun s -> printf s 1000;;
- : (int -> 'a, formatter, unit) format -> 'a
```

Wat.

Where is the magic?

```
# printf ;;
- : ('a, formatter, unit) format -> 'a
# printf "%d sabords!" 1000;;
1000 sabords!
# printf "%d sabords!" 10.5;;
Error: This expression has type float but
an expression was expected of type int
# printf "%d sabords!";;
- : int -> unit
# fun s -> printf s 1000;;
- : (int -> 'a, formatter, unit) format -> 'a
```

Wat.

Where is the magic?

```
# printf ;;
- : ('a, formatter, unit) format -> 'a
# printf "%d sabords!" 1000;;
1000 sabords!
# printf "%d sabords!" 10.5;;
Error: This expression has type float but
an expression was expected of type int
# printf "%d sabords!";;
- : int -> unit
# fun s -> printf s 1000;;
- : (int -> 'a, formatter, unit) format -> 'a
```

Wat.

Where is the magic?

```
# printf ;;
- : ('a, formatter, unit) format -> 'a
# printf "%d sabords!" 1000;;
1000 sabords!
# printf "%d sabords!" 10.5;;
Error: This expression has type float but
an expression was expected of type int
# printf "%d sabords!";;
- : int -> unit
# fun s -> printf s 1000;;
- : (int -> 'a, formatter, unit) format -> 'a
```

Wat.

An interlude in Prolog

Very short introduction to prolog

Prolog is an (untyped) *logic* programming language (more precisely, first order logic). If you have the occasion, learn it, it's very fun.

```
?- length([1, 3, 6], L).
L = 3.

?- append([3], [2, 1], Z).
Z = [3, 2, 1].

?- append([3], X, [3, 4, 5]).
X = [4, 5].

?- append([H], T, Z).
Z = [H|T].
```

Very short introduction to prolog

Prolog is an (untyped) *logic* programming language (more precisely, first order logic). If you have the occasion, learn it, it's very fun.

```
?- length([1, 3, 6], L).
L = 3.
?- append([3], [2, 1], Z).
Z = [3, 2, 1].
?- append([3], X, [3, 4, 5]).
X = [4, 5].
?- append([H], T, Z).
Z = [H|T].
```

Very short introduction to prolog

Prolog is an (untyped) *logic* programming language (more precisely, first order logic). If you have the occasion, learn it, it's very fun.

```
?- length([1, 3, 6], L).
L = 3.

?- append([3], [2, 1], Z).
Z = [3, 2, 1].

?- append([3], X, [3, 4, 5]).
X = [4, 5].

?- append([H], T, Z).
Z = [H|T].
```

Very short introduction to prolog

Prolog is an (untyped) *logic* programming language (more precisely, first order logic). If you have the occasion, learn it, it's very fun.

```
?- length([1, 3, 6], L).
L = 3.
?- append([3], [2, 1], Z).
Z = [3, 2, 1].
?- append([3], X, [3, 4, 5]).
X = [4, 5].
?- append([H], T, Z).
Z = [H|T].
```

Very short introduction to prolog

Prolog is an (untyped) *logic* programming language (more precisely, first order logic). If you have the occasion, learn it, it's very fun.

```
?- length([1, 3, 6], L).
L = 3.
?- append([3], [2, 1], Z).
Z = [3, 2, 1].
?- append([3], X, [3, 4, 5]).
X = [4, 5].
?- append([H], T, Z).
Z = [H|T].
```

Difference lists

You can keep the tail of a list as a variable: `[a,b,c,d|T]`

Then, appending is easy: you just need to unify T.

```
?- L = [a,b,c,d|T], T = [1,2,3].
L = [a, b, c, d, 1, 2, 3]
```

With difference lists, concatenation is O(1).

A *difference list* is a pair or a list and its tail: `[a,b,c,d|T]-T`.

Unification

Prolog shows us that we can compute on lists with unification.

Hindley-Milner type systems are great at doing unification.

Unification

Prolog shows us that we can compute on lists with unification.

Hindley-Milner type systems are great at doing unification.

Greenspun's Tenth Rule

Any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp.

Unification

Prolog shows us that we can compute on lists with unification.

Hindley-Milner type systems are great at doing unification.

The prolog rule of type systems

Any sufficiently complicated type system contains an ad hoc slow implementation of half of prolog.

Prolog in the OCaml type system

Type level lists

`'ty` is the type level list.

`'var` is the unification variable at the tail.

```
type ('ty, 'var) t =
  | Nil : ('var, 'var) t
  | Cons :
    'a * ('ty, 'var) t -> ('a -> 'ty, 'var) t
```

We count with the number of arrows!

```
# Cons(1,Nil);;
- : (int -> 'v, 'v) t
# Cons("foo", Cons(false,Nil));;
- : (string -> bool -> 'v, 'v) t
```

'ty is the type level list.

'var is the unification variable at the tail.

```
type ('ty, 'var) t =
  | Nil : ('var, 'var) t
  | Cons :
    'a * ('ty, 'var) t -> ('a -> 'ty, 'var) t
```

We count with the number of arrows!

```
# Cons(1,Nil);;
- : (int -> 'v, 'v) t
# Cons("foo", Cons(false,Nil));;
- : (string -> bool -> 'v, 'v) t
```

Type level lists

`'ty` is the type level list.

`'var` is the unification variable at the tail.

```
type ('ty, 'var) t =
  | Nil : ('var, 'var) t
  | Cons :
    'a * ('ty, 'var) t -> ('a -> 'ty, 'var) t
```

We count with the number of arrows!

```
# Cons(1,Nil);;
- : (int -> 'v, 'v) t
# Cons("foo", Cons(false,Nil));;
- : (string -> bool -> 'v, 'v) t
```

Type level lists

`'ty` is the type level list.

`'var` is the unification variable at the tail.

```
type ('ty, 'var) t =
  | Nil : ('var, 'var) t
  | Cons :
    'a * ('ty, 'var) t -> ('a -> 'ty, 'var) t
```

We count with the number of arrows!

```
# Cons(1,Nil);;
- : (int -> 'v, 'v) t
# Cons("foo", Cons(false,Nil));;
- : (string -> bool -> 'v, 'v) t
```

```
# let one x = Cons (x, Nil) ;;
val one : 'a -> ('a -> 'v, 'v) t
```

$$'\mathrm{ty} = \alpha \to '\mathrm{v}$$

$$'\mathrm{ty} - '\mathrm{v} = \alpha$$

Terrible arithmetic for apprentice type magicians

```
# let one x = Cons (x, Nil) ;;
val one : 'a -> ('a -> 'v, 'v) t
```

$$\text{'ty} = \alpha \rightarrow \text{'v}$$

$$\text{'ty} - \text{'v} = \alpha$$

Terrible arithmetic for apprentice type magicians

```
# let one x = Cons (x, Nil) ;;
val one : 'a -> ('a -> 'v, 'v) t
```

$$\text{'ty} = \alpha \to \text{'v}$$

$$\text{'ty} - \text{'v} = \alpha$$

Append for difference lists

```
# Cons("foo", Cons(false,Nil));;
- : (string -> bool -> 'v1, 'v1) t
# Cons(1,Nil);;
- : (int -> 'v2, 'v2) t
# Cons("foo", Cons(false, Cons(1,Nil)));;
- : (string -> bool -> int -> 'v3, 'v3) t
```

We replace `'v1` in `string -> bool -> 'v1` by `int -> 'v2`.

We can deduce the type for append:

```
val append:
  ('ty1,'ty2) t -> ('ty2,'v) t -> ('ty1,'v) t
```

Append for difference lists

```
# Cons("foo", Cons(false,Nil));;
- : (string -> bool -> 'v1, 'v1) t
# Cons(1,Nil);;
- : (int -> 'v2, 'v2) t
# Cons("foo", Cons(false, Cons(1,Nil)));;
- : (string -> bool -> int -> 'v3, 'v3) t
```

We replace `'v1` in `string -> bool -> 'v1` by `int -> 'v2`.

We can deduce the type for append:

```
val append:
  ('ty1,'ty2) t -> ('ty2,'v) t -> ('ty1,'v) t
```

Append for difference lists

```
val append:
  ('ty1,'ty2) t -> ('ty2,'v) t -> ('ty1,'v) t
```

$$
\begin{aligned}
& \text{'ty}_1 - \text{'ty}_2 \\
+ & \text{'ty}_2 - \text{'v} \\
= & \text{'ty}_1 - \text{'v}
\end{aligned}
$$

Append for difference lists – Implementation

```
let rec append
  : type ty1 ty2 v.
    (ty1, ty2) t ->
    (ty2, v  ) t ->
    (ty1, v  ) t
  = fun l1 l2 -> match l1 with
  | Nil -> l2
  | Cons (h, t) -> Cons (h, append t l2)
```

The other lists functions are left as an exercise for the audience.

Append for difference lists – Implementation

```
let rec append
  : type ty1 ty2 v.
    (ty1, ty2) t ->
    (ty2, v  ) t ->
    (ty1, v  ) t
  = fun l1 l2 -> match l1 with
  | Nil -> l2
  | Cons (h, t) -> Cons (h, append t l2)
```

The other lists functions are left as an exercise for the audience.

Back to printf

What is a format

There is a bit of compiler magic in OCaml to recognize formats:

```
# ("%s | %s" : _ format) ;;
- : (string -> string -> 'a, 'b, 'a) format
```

This type looks like our new list datatype!

The format datatype

```
type ('ty,'v) t =
| End : ('v,'v) t
| Constant : string * ('ty,'v) t -> ('ty,'v) t
| Hole : ('ty, 'v) t -> (string -> 'ty, 'v) t

# Hole (Constant (" | ", Hole End)) ;;
- : (string -> string -> 'v, 'v) format
```

The format datatype

```
type ('ty,'v) t =
| End : ('v,'v) t
| Constant : string * ('ty,'v) t -> ('ty,'v) t
| Hole : ('ty, 'v) t -> (string -> 'ty, 'v) t

# Hole (Constant (" | ", Hole End)) ;;
- : (string -> string -> 'v, 'v) format
```

The format datatype

```
type ('ty,'v) t =
| End : ('v,'v) t
| Constant : string * ('ty,'v) t -> ('ty,'v) t
| Hole : ('ty, 'v) t -> (string -> 'ty, 'v) t

# Hole (Constant (" | ", Hole End)) ;;
- : (string -> string -> 'v, 'v) format
```

The format datatype

```
type ('ty,'v) t =
| End : ('v,'v) t
| Constant : string * ('ty,'v) t -> ('ty,'v) t
| Hole : ('ty, 'v) t -> (string -> 'ty, 'v) t

# Hole (Constant (" | ", Hole End)) ;;
- : (string -> string -> 'v, 'v) format
```

printf – Implementation

We want to implement `printf`

```
val printf: ('ty, string) t -> 'ty}
```

But the number of argument could be arbitrary!

Instead, we implement first by continuation:

```
val kprintf: (string -> 'v) -> ('ty, 'v) format -> 'ty
```

This is easy to write, you can try it :)

The whole implementation is included in the supported code.

You might be wondering: is this *really* how printf works?

```
# ("%s | %s" : _ format) ;;
- : (string -> string -> 'a, 'b, 'a) format =
CamlinternalFormatBasics.(Format(
 String (No_padding, String_literal (" | ",
   String (No_padding, End_of_format))),
 "%s | %s"))
```

Originally written in 1996 by Pierre Weis (GADT didn't even existed!?)[4]

Rewritten in 2014 by Benoit Vaugon using GADTs. The actual implementation is a lot more complex than our toy example.

[4] It was full of `Obj.magic`

You might be wondering: is this *really* how printf works?

```
# ("%s | %s" : _ format) ;;
- : (string -> string -> 'a, 'b, 'a) format =
CamlinternalFormatBasics.(Format(
 String (No_padding, String_literal (" | ",
   String (No_padding, End_of_format))),
 "%s | %s"))
```

Originally written in 1996 by Pierre Weis (GADT didn't even existed!?)[4]

Rewritten in 2014 by Benoit Vaugon using GADTs. The actual implementation is a lot more complex than our toy example.

[4] It was full of `Obj.magic`

Real World Printf

You might be wondering: is this *really* how printf works?

```
# ("%s | %s" : _ format) ;;
- : (string -> string -> 'a, 'b, 'a) format =
CamlinternalFormatBasics.(Format(
 String (No_padding, String_literal (" | ",
   String (No_padding, End_of_format))),
 "%s | %s"))
```

Originally written in 1996 by Pierre Weis (GADT didn't even existed!?)[4]

Rewritten in 2014 by Benoit Vaugon using GADTs. The actual implementation is a lot more complex than our toy example.

[4] It was full of `Obj.magic`

Wrapping up

- We can use unification to compute in types.
- GADTs allow us to define such datatype relatively easily.
- Prolog is fun.
- We can use GADT for useful things.
- You will only understand this by practice.

Wrapping up

- We can use unification to compute in types.
- GADTs allow us to define such datatype relatively easily.
- Prolog is fun.
- We can use GADT for useful things.
- You will only understand this by practice.

Real World GADTs

`Bigarray`	Controlling memory layout
`Format`	Type level lists for Printf
`Hmap`	Heterogeneous maps
`SLAP`	Linear algebra with statically checked sizes
	Many type-safe DSLs: URL routing, GraphQL APIs, Typed regular expressions, SMT terms, Organize devices for unikernels
	...

Real World Printf

Commit e0b000527 by Gabriel Scherer about Printf

[..] The short summary is [...] that proving things by writing GADT functions in OCaml reveals that Coq's Ltac is a miracle of usability.

Questions ?

Code at `https://gabriel.radanne.net/talks`

Detailed blog post on `https://drup.github.io/2016/08/02/difflists/`

Troubles in GADT paradise

For technical reasons, our GADT type is not covariant, which mean we don't enjoy the relaxed value restriction.

```
# append
  (Cons (1, Cons ("bla", Nil)))
  (Cons (2., Nil))
- : (int -> string -> float -> '_v, '_v) t
```

This means formats are a bit annoying to use in a functional way.

printf – Implementation

We want to implement `printf: ('ty, string) t -> 'ty`.

printf – Implementation

We want to implement `printf: ('ty, string) t -> 'ty`.

```
# let x = Hole (Constant (" | ", Hole End)) ;;
val x : (string -> string -> 'v, 'v) format
# printf x;;
- : string -> string -> string
```

printf – Implementation

We want to implement `printf: ('ty, string) t -> 'ty`.

```
let rec printf
: type ty v. (ty,v) t -> ty
= fun k -> function
  | End -> ""
  | Constant (const, fmt) ->
    const ^ printf fmt (* oups *)
  | Hole fmt ->
    fun x -> x ^ printf fmt (* oups *)
```

Recursive calls to printf might have more arguments. That doesn't work.

printf – Implementation

We want to implement `printf: ('ty, string) t -> 'ty`.

Instead, we are going to implement by continuation:

```
val kprintf:
  (string -> 'v) -> ('ty, 'v) format -> 'ty
```

printf – Implementation

We want to implement `printf: ('ty, string) t -> 'ty`.

```
let rec kprintf
: type ty v. (string -> v) -> (ty,v) t -> ty
= fun k -> function
  | End -> k ""
  | Constant (const, fmt) ->
    kprintf (fun str -> k (const ^ str)) fmt
  | Hole fmt ->
    let f s =
      kprintf (fun str -> k (s ^ str)) fmt
    in f
```

printf – Implementation

We want to implement `printf: ('ty, string) t -> 'ty`.

```
let rec kprintf
: type ty v. (string -> v) -> (ty,v) t -> ty
= fun k -> function
  | End -> k ""
  | Constant (const, fmt) ->
    kprintf (fun str -> k (const ^ str)) fmt
  | Hole fmt ->
    let f s =
      kprintf (fun str -> k (s ^ str)) fmt
    in f
let printf fmt = kprintf (fun x -> x) fmt
```

Balanced parens

```
type zero = Zero
type 'a succ = Succ
type _ t =
  | End : zero t
  | R : 'a t -> 'a succ t
  | L : 'a succ t -> 'a t
type start = Start of zero t

(* (()()) *)
let x = Start (L (L (R (L (R (R End)))))) ;;
```

We can encode any FSA with an arbitrary (finite) number of registers.

Note: not a minsky machine: no conditional jumps.