

# Generalized Algebraic Data Types in OCaml

## Die Sprache **L-If**

$Atom ::= true \mid false \mid 0..9^+$

$Expr ::= Atom \mid \text{if } Expr \text{ then } Expr \text{ else } Expr$

## Gültige Programme

```
true
```

```
if true then 42 else 0
```

```
if true then  
  if true then 42 else 0  
else  
  0
```

## Ungültige Programme

```
if 0 then true else false
```

```
if true then 0 else false
```

```
if true then  
  if true then 42 else 0  
else  
  false
```

# In einer Welt ohne GADTs: L-ADT-If

```
type atom =  
  | Bool of bool  
  | Int  of int  
  
type expr =  
  | Atom of atom  
  | If   of expr * expr * expr
```

Syntaxbaum für L-ADT-If

```
let rec eval: expr → atom = function
| Atom a → a
| If (c, i, e) → begin match eval c with
    | Bool true → eval i
    | Bool false → eval e
    | _ → failwith "expected bool!"
end
```

*Evaluationsfunktion für L-ADT-If*

# Probleme von L-ADT-If

```
eval (If  
      (Atom (Bool true),  
        (Atom (Int 42)),  
        (Atom (Int 0))))
```

```
- : atom = Atom (Int 42)
```

```
eval (If  
      (Atom (Int 42),  
        (Atom (Bool false)),  
        (Atom (Int 0))))
```

```
Exception: Failure "need bool!"
```

*Beispiele in L-ADT-If*

- Ungültige Programmdefinition möglich
- Laufzeitfehler im Interpreter
- Zweige können verschiedene Typen haben

# In einer Welt mit GADTs: L-GADT-If

```
type _ atom =  
  | Bool : bool → bool atom  
  | Int  : int  → int  atom  
  
type _ expr =  
  | Atom : 'a atom → 'a expr  
  | If   : bool expr * 'a expr * 'a expr → 'a expr
```

*Syntaxbaum für L-GADT-If*

# ADTs vs GADTs

```
type atom =  
  | Bool : bool → atom  
  | Int  : int  → atom
```

*L-ADT-If atom*

```
type _ atom =  
  | Bool : bool → bool atom  
  | Int  : int  → int  atom
```

*L-GADT-If atom*

Konstrukturen eines GADTs können *verschiedene* Typen annehmen, während bei ADTs alle Konstruktoren den *selben* Typ haben.



```
let rec eval : .. = function
  | Atom (Bool b) → b
  | Atom (Int i)  → i
  | If (c, i, e)  → if eval c then eval i else eval e
```

*Evaluationsfunktion für L-GADT-If*

# Die Vorteile von L-GADT-If

```
eval (If  
      (Atom (Bool true),  
        (Atom (Int 42)),  
        (Atom (Int 0))))
```

```
eval (If  
      (Atom (Int 42),  
        (Atom (Bool false)),  
        (Atom (Int 0))))
```

```
- : int = 42
```

*Beispiele in L-GADT-If*

- Keine ungültigen Programmdefinitionen
- Keine Laufzeitfehler im Interpreter
- Exhaustive Patternmatching

# Local Abstrakte Typen

```
let rec eval (type a) (e : a expr) : a = match e with
| Atom (Bool b) → b
| Atom (Int i)  → i
| If (c, i, e)  → if eval c then eval i else eval e
```

*Evaluationsfunktion mit Lokal Abstrakten Typen für L-GADT-If*

# Polymorphe Rekursion

```
let rec eval : 'a. 'a expr → 'a =  
  fun (type a) (e : a expr) : a → match e with  
    | Atom (Bool b) → b  
    | Atom (Int i)  → i  
    | If (c, i, e)   → if eval c then eval i else eval e
```

*Evaluationsfunktion mit L.A.T. und polymorpher Rekursion für L-GADT-If*

```
let rec eval : type a. a expr → a = function
| Atom (Bool b) → b
| Atom (Int i) → i
| If (c, i, e) → if eval c then eval i else eval e
```

*Evaluationsfunktion für L-GADT-If*

# Unterschiedliche Rückgabewerte

```
type (_, _) mode =  
  | Unsafe : ('a, 'a) mode  
  | Option : ('a, 'a option) mode
```

```
let first : type a r. a list → (a, r) mode → r =  
  fun lst mode → match lst with  
    | hd :: tl → (match mode with  
      | Unsafe → hd  
      | Option → Some hd)  
    | [ ] → (match mode with  
      | Unsafe → failwith "list is empty"  
      | Option → None)
```

Funktion `first` mit unterschiedlichen Rückgabewerten je nach `mode`

# Existenzielle Typen

```
type stringable =  
  Stringable : {  
    item: 'a;  
    to_string: 'a → string  
  } → stringable
```

```
let print (Stringable s) =  
  print_endline (s.to_string s.item)
```

Existenzieller Typ `stringable` mit `print` Funktion

# In einer Nusschale

- GADTs sind eine Erweiterung von ADTs, die es erlaubt, dass Variants verschiedene Typen, basierend auf jeweils eigenen Typvariablen, annehmen können
- Beim Patternmatching extrahiert der Compiler über GADTs mehr Informationen und kann somit mehr Fälle eliminieren
- Mit GADTs lassen sich einige interessante Konzepte realisieren, wie Existenzielle Typen, Funktionen mit verschiedenen Rückgabewerten und im generellen ausdrucksstärkere Typdefinitionen
- Allerdings wird mit GADTs die Typinferenz unentscheidbar, weshalb Typannotationen benötigt werden, zudem benötigt es neue Konzepte um rekursive Funktionen zu realisieren



# Folien

<https://mari-w.github.io/ocaml-gadts/>

## Literatur

- Real World OCaml: GADTs  
Yaron Minsky, Anil Madhavapeddy 2021
- Detecting use-cases for GADTs in OCaml  
Mads Hartmann 2015
- Stanford CS242: Programming Languages  
Will Crichton 2019