Towards proved programs
Part I: Introducing GADTs

MPRI 2-4-2

(Version Fri Jan 16 13:01:50 CET 2015)

Yann **Régis-Gianas**
yrg@pps.univ-paris-diderot.fr

Paris 7 - PPS
INRIA - $\pi.r^2$

January 16, 2015

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

GADTs in other type systems

GADTs through a motivating example

Let us have a second look to the `List.hd` function...

Algebraic data types : data constructors

Consider an `OCaml` type definition for list of integers:

```
type ilist =
  | Nil
  | Cons of int × ilist
```

- `Nil` and `Cons` are **data constructors**.
- Only way of obtaining values of type `ilist`.
- Data constructors **tag** specific tuples of values.
- They act as **dynamic witnesses** for these values.

Algebraic data types : pattern matching

From the point of view of the type system, an empty list and a nonempty list are **indistinguishable**: both have type `ilist`.

Yet, thanks to the tag, a **dynamic test** can help recover locally the exact nature of the list:

```
let hd l =
  match l with
  | Nil → (* The list "l" is empty.  *) failwith "hd"
  | Cons (x, _) → (* The list "l" is not empty.  *) x
```

A safer head extraction?

Is it possible to change the type definition for lists to obtain a version of `hd` that never fails?

A trick in vanilla ML : Phantom types [Leijen and Meijer, 1999]

Type-level programming trick: Phantom types

Add an extra type parameter to a type definition, free in the body of this definition, so that it can be freely instantiated to transmit piece of static information to the type-checker.

In other words, the type parameter α of a value $\mathtt{v}$ of phantom type α $\mathtt{t}$ will not represent the type of a component of $\mathtt{v}$ but a static property attached to $\mathtt{v}$.

A phantom type for emptiness

Let us add such an extra type parameter to the type of lists:

type α plist $=$ ilist

Emptiness of lists can be encoded at the level of types using two fresh distinct type constructors:

type empty
type nonempty

Smart constructors to define a phantom type

By turning this type into an abstract data type [Liskov and Zilles, 1974], we can specialize the type of the data constructors for lists:

```
module List : sig
  type ilist = private Nil | Cons of int × ilist
  type α plist = ilist
  val nil : empty plist
  val cons : int → α plist → nonempty plist
  val hd : nonempty plist → int
end = struct
  type ilist = Nil | Cons of int × ilist
  type α plist = ilist
  let nil = Nil
  let cons x xs = Cons (x, xs)
  let hd = function Cons (x, _) → x | Nil → assert false
end
```

A safer (?) head extraction

```
let hd : nonempty List.plist → int = function
  | Nil → assert false
  | Cons (x, _) → x
```

Thanks to this phantom type, some ill-formed application of `hd` are rejected by the type-checker.

Indeed, the term `hd nil` is clearly ill-typed since `empty` and `nonempty` are two incompatible types.

On the contrary, the term `hd (cons 1 nil)` is well-formed.

Limit of phantom types

```
let totalHd : type a. a plist → int = function
  | Nil → 42
  | (Cons _) as l → hd l
```

Even if the use of `hd` is perfectly sound, the type-checker rejects it because it cannot deduce from the pattern matching that the only possible type assignment for the type variable `a` is `nonempty`.

This is where **Generalized Algebraic Data Types** enter the scene.

Generalized Algebraic Data Types

A type-checker for Generalized Algebraic Data Types (GADTs) overcomes the over-mentioned limitation through **an automatic local refinement of the typing context in each branch of the pattern matching**.

Our first GADT

```
(* This code fragment is accepted by ocaml >= 4.00 *)
type α glist =
  | Nil : empty glist
  | Cons : int × α glist → nonempty glist
```

Like smart constructors of phantom types, this declaration **restricts the type of constructed values with respect to the data constructors they are built with**.

In addition, the fact that specific types are now attached to data constructors and not to smart constructors, which are regular values, yields **a more precise typing of pattern matching branches**.

Our first pattern matching over a GADT

The type-checker can now **deduce that some cases are absurd** in a particular typing context. For instance, in the following definition of `hd`:

```
let hd : nonempty glist → int = function
  | Cons (x, _) → x
```

. . . the type-checker will not complain anymore that the pattern matching is not exhaustive.

Indeed, it is capable of proving that the unspecified case corresponding to the pattern `Nil` is impossible given that such a pattern would only capture values of type `empty glist`, which is incompatible with `nonempty glist`.

Automatic type refinement in action

Besides, a type-checker for GADTs is able to **refine the type of** `l` in the second case of this pattern matching:

```
let totalHd : type a. a glist → int = function
  | Nil → 42
  | (Cons _ ) as l → hd l
```

In the body of the last branch, the type-checker knows that the **term** `l` **has both the type** `a glist` **and the type** `nonempty glist` because `l` has been built by application of the data constructor `Cons`. The second type is enough to accept the application of `hd` on `l`.

Where does the magic lie?

Which technical device is used by the type-checker to assign several types to one particular term?

Type equalities

We can morally reformulate our previous GADT definition by attaching a **type equality** to each data constructor:

type α glist =
  | Nil : $[\alpha = \text{empty}]\ \alpha$ glist
  | Cons : $[\alpha = \text{nonempty}]$ int $\times\ \beta$ glist $\to\ \alpha$ glist

(These types are similar to constrained type schemes of HM(X).)

Local type equalities

- In the right-hand side of a pattern matching branch corresponding to a particular data constructor, the associated type equalities are **implicitly assumed** by the type-checker.
- A **conversion rule** allows type assignment *modulo* these type equalities.

Type-checking `totalHd`

```
let totalHd : type a. a glist → int = function
  | Nil → 42
  | (Cons _) as l → hd l
    (* Here, a = nonempty.  *)
    (* As Γ ⊢ l :  a glist *)
    (* and a = nonempty ⊨ a glist = nonempty glist, *)
    (* by conversion, Γ ⊢ l :  nonempty glist *)
```

Origins of GADTs

Generalized Algebraic Data types have been first introduced independently by Hongwei Xi, Chiyan Chen and Gang Chen [Xi et al., 2003] and by James Cheney and Ralf Hinze [Cheney and Hinze, 2003].

They draw inspiration from **inductive definitions** of Coq. Yet, as there is no (direct) dependency between values and types, GADTs have features that cannot be obtained in the Calculus of Inductive Constructions.

Exercise

Consider the following two type definitions:

```
type zero
```

These data constructors are useful to encode type-level natural numbers using Peano's numeration.

① Write a GADT definition that enables the type-checker to keep track of the length of lists.

Exercises

② Consider the following recursive function:

```
let rec sum l1 l2 =
  match l1, l2 with
  | [], [] → []
  | x :: xs, y :: ys → (x + y) :: sum xs ys
  | _ → failwith "sum"
```

Rewrite it using your definition of lists. Why is the type of your function more accurate than the usual one?

Exercises

③ Examine the following function:

```
let rec append l1 l2 =
  match l1 with
  | [] → l2
  | x :: xs → x :: (append xs l2)
```

Is it possible to obtain a well-typed version of this function using your definition of lists?

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

- Syntax
- Operational semantics
- Type System
- Metatheory
- Implementation of a type checker for MLGI

Applications

GADTs in other type systems

MLGI

Now, a formal presentation of ML extended with implicit GADTs (MLGI).

The use of the adjective "implicit" will be justified by the existence of an **implicit conversion rule** in the type system. A version of this language with explicit conversion will be introduced in the lecture about type reconstruction.

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

- Syntax
- Operational semantics
- Type System
- Metatheory
- Implementation of a type checker for MLGI

Applications

GADTs in other type systems

Syntax of MLGI

`Terms` t	$::=$	
Variable	$\mid$	x
Function	$\mid$	$\lambda x.t$
Function application	$\mid$	$t\ t$
Local definition	$\mid$	$\mathsf{let}\, x\ =\ t\ \mathsf{in}\ t$
Fixpoint	$\mid$	$\mu x.t$
Data constructor application	$\mid$	$K\,(t, \ldots, t)$
Case analysis	$\mid$	$\mathsf{match}\, t\, \mathsf{with}\, \overline{c}$

Syntax of MLGI (continued)

Two new syntactic categories for pattern matching: clauses and patterns.

$$\texttt{Clauses} \quad c \quad ::= \quad p \Rightarrow t$$

$$\texttt{Patterns} \quad p \quad ::= \quad K\ (\overline{x})$$

A **clause** c, or "branch" or "case", is a pair written $p \Rightarrow t$ of a pattern p and a term, called its body.

A **pattern** describes a shape that may or may not match some scrutinee.

Our syntax defines very basic patterns, called *flat patterns*, that only allow for a discrimination on the tag of a value. More complicated patterns can be defined (see a forthcoming exercise) but flat patterns will be expressive enough for us.

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

- Syntax
- Operational semantics
- Type System
- Metatheory
- Implementation of a type checker for MLGI

Applications

GADTs in other type systems

Values

`Values`	$v ::=$	
Function		$\lambda x.t$
Data		$K\,(v_1, \ldots, v_n)$

The values are extended with **tagged tuple of values**.

Evaluation context

`Evaluation context`	$\mathbb{E} ::=$	
Hole		$[]$
Left hand side of an application	$\mid$	$\mathbb{E}\ t$
Right hand side of an application	$\mid$	$v\ \mathbb{E}$
Left hand side of a let	$\mid$	$\mathsf{let}\, x\ =\ \mathbb{E}\ \mathsf{in}\ t$
Right hand side of a let	$\mid$	$\mathsf{let}\, x\ =\ v\ \mathsf{in}\ \mathbb{E}$
Arguments of data constructors	$\mid$	$K\,(v_1, \ldots, v_n, \mathbb{E}, t_1, \ldots, t_k)$
Scrutinee evaluation	$\mid$	$\mathsf{match}\ \mathbb{E}\ \mathsf{with}\ \overline{c}$

As expected, we force the evaluation of the scrutinee before the evaluation of the branches

Reduction rules

$$\text{(E-Context)} \quad \mathbb{E}\,[t] \to \mathbb{E}\,[t'] \quad \text{if } t \to t'$$

$$\text{(E-Beta)} \quad (\lambda x.t)v \to [x \mapsto v]t$$

$$\text{(E-Let)} \quad \text{let}\, x = v \,\text{in}\, t \to [x \mapsto v]t$$

$$\text{(E-Fix)} \quad \mu x.t \to [x \mapsto \mu x.t]t$$

$$\text{(E-Match)} \quad \text{match}\, K\,(v_1, \ldots, v_n)\,\text{with}\; K\;\; (x_1, \ldots, x_n) \Rightarrow t \mid \overline{c} \to [x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]t$$

$$\text{(E-NoMatch)} \quad \text{match}\, K'\,(v_1, \ldots, v_m)\,\text{with}\; K\;\; (x_1, \ldots, x_n) \Rightarrow t \mid \overline{c} \to \text{match}\, K'\,(v_1, \ldots, v_m)\,\text{with}\; \overline{c} \quad \text{if } K \neq K'$$

Stuck terms

Misplaced tagged value If a term contains an application whose left-hand side is a tagged value, this term is stuck. Hence, a type system should use a type constructor different from the arrow for this kind of values.

Misplaced function Dually, a function cannot be a scrutinee. Notice that this is not true if the syntax of patterns would have included a case for wildcard or variable.

Exhausted cases When a scrutinee has not been captured by any of the cases of a pattern matching, the evaluation is stuck on a term of the form "match $K\,(v_1, \ldots, v_m)$ with $\emptyset$". Hence, a type-checker must ensure that pattern matchings are exhaustive.

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

- Syntax
- Operational semantics
- Type System
- Metatheory
- Implementation of a type checker for MLGI

Applications

GADTs in other type systems

Syntax for types

$$
\begin{array}{rrl}
\texttt{Types} & \tau ::= & \\
\textit{Type variable} & & \alpha \\
\textit{Function type} & & \mid \tau \to \tau \\
\textbf{Type application} & & \mid \varepsilon\, \overline{\tau}\, \overline{\tau} \\
\texttt{Type schemes} & \sigma ::= & \forall \overline{\alpha}.\tau
\end{array}
$$

The type algebra now contains a case for algebraic data types formed by application of a type constructor ε to type parameters.

To simplify formalization, we split type parameters of ε into two disjoint sets : *regular* and *generalized* type parameters. If the second set of type parameters is empty, then ε is a regular algebraic data type. Otherwise, ε is a generalized algebraic data type. [1]

[1] Notice that, contrary to the `OCaml`'s convention, we write type application using a postfix notation.

Specification of data constructors

We assume that algebraic data type constructors are introduced by toplevel declarations consisting of a name ε for the type constructor and a fixed set of data constructors declared using the following syntax:

$$K :: \forall \overline{\beta}\overline{\alpha}.\ \tau_1 \times \ldots \times \tau_m \to \varepsilon\, \overline{\alpha}\, \overline{\tau}$$

with $\overline{\beta} \# \overline{\alpha}$, $\overline{\beta} \in \mathbf{ftv}\,(\overline{\tau})$ and $\overline{\alpha}$ are all distinct.

We write $K \preceq \sigma$ as a shortcut for

$$\forall \overline{\beta}\overline{\alpha}.\ \tau_1 \times \ldots \times \tau_m \to \varepsilon\, \overline{\alpha}\, \overline{\tau} \preceq \sigma$$

GADTs includes ADTs

Observe that, if $\overline{\beta}$ is empty and the set of generalized type parameters is also empty, this form of declaration degenerates into:

$$K :: \forall \overline{\alpha}.\ \tau_1 \times \ldots \times \tau_m \to \varepsilon\, \overline{\alpha}$$

which indeed corresponds to a declaration for a data constructor of a regular algebraic data type.

Example

Here is two data constructor declarations for polymorphic lists:

$$\begin{array}{rcl} \mathit{Nil} & :: & \forall \alpha.\mathrm{list}\,\alpha \\ \mathit{Cons} & :: & \forall \alpha.\alpha \times \mathrm{list}\,\alpha \to \mathrm{list}\,\alpha \end{array}$$

GADTs includes iso-existential data types

The type variables $\overline{\beta}$ are said to be local because when the set of generalized type parameters is empty, the $\overline{\beta}$ do not escape through the output type. As a consequence, they can be used to implement iso-existential types [Läufer and Odersky, 1992].

Example

The following data constructor declaration is well-suited to type closures for functions of type $\alpha_1 \to \alpha_2$:

$$\mathit{Closure} \quad :: \quad \forall \beta \alpha_1 \alpha_2 . \beta \times (\beta \to \alpha_1 \to \alpha_2) \to \mathrm{closure}\, \alpha_1\, \alpha_2$$

Generalized ADTs

If we exploit the full expressiveness of the syntax of data constructor declarations, we get GADTs.

Example

Polymorphic lists indexed by a type-level natural number encoding their length are introduced by the following data constructors:

$$\begin{aligned} \mathit{Nil} &:: \forall\alpha.\mathrm{list}\,\alpha\,\mathrm{zero} \\ \mathit{Cons} &:: \forall\beta\alpha.\alpha \times \mathrm{list}\,\alpha\,\beta \to \mathrm{list}\,\alpha\,(\mathrm{succ}\,\beta) \end{aligned}$$

list is a GADT whose first type parameter is regular while the second is generalized.

Syntax for type equalities

As coined earlier, the type system for GADTs will make use of a set of local type equalities, written as a conjunction of equalities between types:

$$\texttt{Equation systems} \quad E ::= \mathsf{true} \mid \tau = \tau \mid E \wedge E$$

Judgments

"$E, \Gamma \vdash t : \tau$", read

> *"Under the local assumptions E and the typing environment Γ, the term t has type τ."*

"$E, \Gamma \vdash t : \sigma$", read

> *"Under the local assumptions E and the typing environment Γ, the term t has type scheme σ."*

"$p : \tau \vdash (\overline{\beta}, E, \Gamma)$", read

> *"The assumption 'the pattern p has type τ' introduces local variables $\overline{\beta}$, local equations E and a local environment Γ."*

Well-typed terms

Definition (Well-typed terms)

A closed term t is well-typed if and only if $E, . \vdash t : \tau$ holds for some **satisfiable** E.

Notice the extra hypothesis: **the type equalities E must be satisfiable**, which means that there must exist a mapping ϕ from type variables to types such that $\phi \models E$.

Type equalities E are called **inconsistent**, which we write "$E \models$ false", if there is no such ϕ.

Rule for data constructor applications

$$\frac{K \preceq \tau_1 \times \ldots \times \tau_n \to \varepsilon\,\overline{\tau}_1\,\overline{\tau}_2 \qquad \forall i \quad E, \Gamma \vdash t_i : \tau_i}{E, \Gamma \vdash K\,(t_1 \ldots t_n) : \varepsilon\,\overline{\tau}_1\,\overline{\tau}_2}\ \texttt{Cstr}$$

The typing rule `Cstr` inlines `Var`, `Inst` and several instances of `App` to check that the type scheme associated to a data constructor K meets the types of the arguments as well as the expected output type.

Conversion rule

$$\textsc{Conv}\ \frac{E, \Gamma \vdash t : \tau_1 \qquad E \models \tau_1 = \tau_2}{E, \Gamma \vdash t : \tau_2}$$

A type conversion can be performed at any point of the term as soon as the equality is implied by the local type equalities, which is written $E \models \tau_1 = \tau_2$ and defined as follows:

$$\frac{}{E_1 \wedge \tau_1 = \tau_2 \wedge E_2 \models \tau_1 = \tau_2} \qquad \frac{E \models \tau_1 = \tau_1' \qquad E \models \tau_2 = \tau_2'}{E \models \tau_1 \to \tau_2 = \tau_1' \to \tau_2'} \qquad \frac{E \models \overline{\tau}_1 = \overline{\tau}_1' \qquad E \models \overline{\tau}_2 = \overline{\tau}_2'}{E \models \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2 = \varepsilon\, \overline{\tau}_1'\, \overline{\tau}_2'}$$

$$\frac{}{E \models \tau = \tau} \qquad \frac{E \models \tau_2 = \tau_1}{E \models \tau_1 = \tau_2} \qquad \frac{E \models \tau_1 = \tau_2 \qquad E \models \tau_2 = \tau_3}{E \models \tau_1 = \tau_3}$$

Rule for pattern matchings

$$\textsc{Case}\ \frac{E, \Gamma \vdash t : \tau_1 \qquad \forall i \quad E, \Gamma \vdash c_i : \tau_1 \to \tau_2}{E, \Gamma \vdash \mathsf{match}\ t\ \mathsf{with}\ c_1 \ldots c_n : \tau_2}$$

The typing rule for pattern matching ensures that the type of the scrutinee is compatible with the type of the pattern of each branch and that the type of the pattern matching is compatible with the type of the body of each branch.

Rule for clauses

$$\frac{\texttt{Clause}\quad p : \varepsilon\,\overline{\tau}_1\,\overline{\tau}_2 \vdash (\overline{\beta}, E', \Gamma') \qquad \overline{\beta} \# \mathbf{ftv}\,(E, \Gamma, \tau_2) \qquad E \wedge E', \Gamma\Gamma' \vdash t : \tau_2}{E, \Gamma \vdash p \Rightarrow t : \varepsilon\,\overline{\tau}_1\,\overline{\tau}_2 \to \tau_2}$$

1. Enforce the type of the scrutinee to be a GADT $\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2$.
2. Extract the local assumptions using $p : \varepsilon\,\overline{\tau}_1\,\overline{\tau}_2 \vdash (\overline{\beta}, E', \Gamma')$.
3. Thread these extra assumptions to type the body of the clause.
4. Check that the local type variables do not escape.

Extraction of type equalities

$$\texttt{Pat}\ \frac{K \preceq \forall \overline{\beta}.\tau_1 \times \ldots \times \tau_n \to \varepsilon\,\overline{\tau}_1\,\overline{\tau} \qquad \overline{\beta} \# \mathbf{ftv}\,(\overline{\tau}_1, \overline{\tau}_2)}{K\ (x_1 \ldots x_n) : \varepsilon\,\overline{\tau}_1\,\overline{\tau}_2 \vdash (\overline{\beta}, \overline{\tau}_2 = \overline{\tau}, (x_1 : \tau_1; \ldots; x_n : \tau_n))}$$

This is the more complex rule: let explain it progressively.

Specialization of Pat to regular ADT

If the data type is a regular algebraic data type, then $\overline{\beta}$ and $\overline{\tau}_2$ are empty and we get:

$$\texttt{Pat-Regular}\quad \frac{K \preceq \tau_1 \times \ldots \times \tau_n \to \varepsilon\,\overline{\tau}_1}{K\ (x_1 \ldots x_n) : \varepsilon\,\overline{\tau}_1 \vdash (\emptyset, \mathsf{true}, (x_1 : \tau_1; \ldots; x_n : \tau_n))}$$

In that specialized version of Pat, the regular type parameters $\overline{\tau}_1$ are used to instantiate the type variables $\overline{\alpha}$ inside the type scheme of K. As a result, the types of the components are instantiated accordingly and can be ascribed to the variable x_i of the pattern.

Specialization of Pat to iso-existential ADT

If the data type is an iso-existential type, then only the $\overline{\tau}_2$ is empty and we get:

$$\texttt{Pat}\ \frac{K \preceq \forall \overline{\beta}.\tau_1 \times \ldots \times \tau_n \to \varepsilon\,\overline{\tau}_1 \qquad \overline{\beta} \# \mathbf{ftv}\,(\overline{\tau}_1)}{K\ (x_1 \ldots x_n) : \varepsilon\,\overline{\tau}_1 \vdash (\overline{\beta}, \mathsf{true}, (x_1 : \tau_1; \ldots; x_n : \tau_n))}$$

In that specialized version of the rule, we can observe that the type variables $\overline{\beta}$ are fresh because they only come from the instantiation of the type scheme of K and cannot occur in $\overline{\tau}_1$.

Back to the general version

$$\texttt{Pat}\quad \frac{K \preceq \forall \overline{\beta}.\tau_1 \times \ldots \times \tau_n \to \varepsilon\, \overline{\tau}_1\, \overline{\tau} \qquad \overline{\beta} \# \mathbf{ftv}\,(\overline{\tau}_1, \overline{\tau}_2)}{K\ (x_1 \ldots x_n) : \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2 \vdash (\overline{\beta}, \overline{\tau}_2 = \overline{\tau}, (x_1 : \tau_1; \ldots; x_n : \tau_n))}$$

We write « $\overline{\tau} = \overline{\tau}_2$ » for the conjunction of equalities between the types of each vectors.

Again, being local, the type variables $\overline{\beta}$ must not escape through the type of the scrutinee.

Generalized type parameters are instantiated with respect to the ordinary type parameters $\overline{\alpha}$ into types $\overline{\tau}_1$. On the contrary, the types $\overline{\tau}$ are not required to be syntactically equal to $\overline{\tau}_2$. Instead, we assume their equalities as a type equality constraint that is exported inside the typing fragment.

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

- Syntax
- Operational semantics
- Type System
- Metatheory
- Implementation of a type checker for MLGI

Applications

GADTs in other type systems

Metatheory

The soundness proof follows the usual scheme. Yet, because of the presence of type equalities in the assumptions of the typing judgment, extra technical lemmas are required.

In addition to the preservation of types through reduction, subject reduction will also **maintain the preservation of the satisfiability** of E under the evaluation.

Key lemma

The following key lemma proves that the local assumptions gained through pattern matching preserves the satisfiability of the constraint E.

Lemma

If $E, \emptyset \vdash K\,(v_1, \ldots, v_n) : \tau$ *and* $K\,(x_1, \ldots, x_n) : \tau \vdash (\overline{\beta}, E', (x_1 : \tau_1, \ldots, x_n : \tau_n))$ *then there exists* E''*, such that :*

(i) $E'' \models E'$

(ii) $\exists \overline{\beta}.\, E \equiv E''$

(iii) $\forall i, E'', . \vdash v_i : \tau_i$

The typing constraint E'' extends E with information about the local type variables $\overline{\beta}$ in a way that is sufficient to entail E'.

Subject reduction

Theorem (Subject reduction)

Let E be satisfiable. If $E, . \vdash t : \tau$ and $t \to t'$ then $E, . \vdash t' : \tau$.

Proof.

By induction on the derivation of $t \to t'$. □

Progress

In addition to the usual stuck terms, we have to take terms of the form:

$$\text{match } v \text{ with } \emptyset$$

into account because they are not values but they are irreducible.

These terms can be easily ruled out by an **exhaustiveness** check.

Theorem (Progress)

If t is well-typed and all pattern matchings of t are exhaustive, then t is either a value or reducible.

Improving progress thanks to dead code elimination

A type-checker for GADTs can do better than the standard syntactical exhaustiveness check.

Even if a pattern matching is not exhaustive with respect to the standard syntactic criterion, a typing argument can prove it is.

Improving progress thanks to dead code elimination

Let us complete every non exhaustive pattern matching with a branch « $p \Rightarrow$ wrong » for each case that is not handled. Let us note $t_\perp$, the image of a term t through this transformation.

The constant wrong represents a runtime error. Morally, this constant should be ill-typed: a well-typed program should never reduce to wrong. Therefore, the only reasonable way for a type-checker to accept an occurrence of this constant in a term is to show that **it is used in a dead code branch**.

Improving progress thanks to dead code elimination

For this reason, we add this rule to the type system:

$$\texttt{Dead}\ \frac{E \models \text{false}}{E, \Gamma \vdash \text{wrong} : \tau}$$

Recall that subject reduction implies that satisfiability of type equalities is preserved by reduction: by contraposition, if type equalities are inconsistent at some point of the term, this term will never be considered by the reduction.

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

- Syntax
- Operational semantics
- Type System
- Metatheory
- Implementation of a type checker for MLGI

Applications

GADTs in other type systems

Type-checking algorithm

The type system we presented is not directly usable in an implementation because it is not syntax directed.

Fortunately, the problem of **checking that a term admit a given type** in MLGI is easy.

On the contrary, the problem of computing a (most general) type for a term is complex and will be studied in the next lecture.

An algorithm based on congruence closure

A type-checker for an explicitly typed ML with GADTs:

1. At every pattern matching branches, compute type equalities E by confronting the type of the scrutinee with the type of the pattern. Two sub-cases:
   - If E is consistent, then convey the local type equalities E along the type-checking of the case branch.
   - If E is inconsistent, then the case branch is dead code. [2]
2. At every point where two types must be equal, decide this type equality *modulo* E using congruence closure [Nelson and Oppen, 1980].

[2] This should raise a typing error, or at least a warning.

An algorithm based on eager application of MGU

Actually, it is not even necessary to convey type equalities: as soon as the type equalities E are known, one can compute a most general unifier ϕ (if E is consistent).

The mgu ϕ can be applied to the environment Γ and to the expected type τ [3] so that every type is normalized with respect to E.

Then, deciding type equalities *modulo* E is replaced with a standard decision procedure for equality.

[3] In practice, this application is not done explicitly but through unification variables.

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

- Embedding domain-specific languages
- Generic programming
- Dependently-typed programming
- Typed intermediate languages

GADTs in other type systems

Successful applications of GADTs

Domain specific languages A tagless interpreter can be written using GADTs to efficiently embed a domain specific language with its own type system inside a general purpose host language.

Dependently-typed programming In some cases, the type schemes of a GADT can encode an algorithmic data structure invariant which is the enforced by the type-checker. Furthermore, a GADT can be used as an inductive predicate and a value of that GADT is then a proof term for that predicate.

Generic programming Providing a dynamic representation of types implemented thanks to a GADT, a lot of daily use combinators, like `fold` or `map`, can be defined once and for all over the structure of the types.

Typed intermediate languages Type-preserving compilation may be helped by GADTs. Indeed, at some point of the compilation chain, the high-level structure of terms may have vanished and be replaced by apparently unrelated low-level concrete data structures. GADTs can sometimes be used to maintain this connection by typing these low-level data structure with high-level related types.

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

- Embedding domain-specific languages
- Generic programming
- Dependently-typed programming
- Typed intermediate languages

GADTs in other type systems

Example : A language of expressions

Algebraic data types are often used to denote abstract syntax trees.

```
(* t ::= 0, 1, ... | π1 t | π2 t | (t, t) *)
type term =
  | Lit of int
  | Pair of term × term
  | Fst of term
  | Snd of term

(* v ::= 0, 1, ... | (v, v) *)
type value =
  | VInt of int
  | VPair of value × value
```

A running example

An interpreter of this language is a straightforward inductive function:

```
let rec eval : term → value = function
  | Lit x → VInt x
  | Pair (t1, t2) → VPair (eval t1, eval t2)
  | Fst t → begin match eval t with
              | VPair (v1, _) → v1
              | _ → failwith "Only pairs can be projected."
            end
  | Snd t → ...
```

How to convince ourselves that well-typed expressions do not go wrong?

Example : A commented interpreter for expressions

Assuming the precondition that input expressions are well-typed, the proof of subject reduction can be interleaved with the code of the interpreter:

```
(*  [eval e] interprets a well-typed expression of type T      *)
(*  into a value of type T, or more formally:                  *)
(*        ∀eT, Pre:"⊢ e : T" ⇒ Post:"⊢ eval e : T"              *)
let rec eval : term → value = function
  | Lit x → VInt x
    (* By inversion of Pre, we have ⊢ Lit x : int              *)
    (* Thus, T = int and, we indeed have, ⊢ VInt x : int       *)
  | Pair (t1, t2) → VPair (eval t1, eval t2)
    (* By inversion of Pre, there exist β1, β2 such that      *)
    (* ⊢ Pair (t1, t2) : β1 × β2, ⊢ t1 : β1, and ⊢ t2 : β2.   *)
    (* Thus, ⊢ eval t1 : β1, and ⊢ eval t2 : β2.              *)
    (* Eventually, we obtain ⊢ (eval t1, eval t2) : β1 × β2.  *)
  | Fst t → ...
  | Snd t → ...
```

Example : A commented interpreter for expressions

```
(* [eval e] interprets a well-typed expression of type T        *)
(* into a value of type T, or more formally:                    *)
(*        ∀eT, Pre:"⊢ e : T" ⇒ Post:"⊢ eval e : T"              *)
let rec eval : term → value = function
  | ...
  | Fst t → begin match eval t with
              | VPair (v1, _) → v1
              | _ → failwith "Only pairs can be projected."
            end
    (* By inversion of Pre, ∃β,⊢ t : T × β                      *)
    (* Then, ⊢ eval t : T × β                                   *)
    (* And by the classification lemma, eval t = (v1, v2)       *)
    (* with ⊢ v1 : T, which is what we expected.                *)
  | Snd t → ...
```

Example : A commented interpreter for expressions

```
let rec eval : term → value = function ...
  | Fst t → begin match eval t with
              | VPair (v1, _) → v1
              | _ → assert false
            end
 (* Thus, by the classification lemma, eval t = (v1, v2)      *)
  | Snd t → ...
```

The underlined step of the proof guarantees that the inner pattern matching always succeed. We can safely replace the error message production with an assertion that this branch cannot be executed (if the precondition holds). As a consequence, this dynamic check performed by the inner pattern matching is **redundant**.

How to encode a predicate using a type?

The type of the interpreter can be enriched in order to relate the input and the output by **sharing a type variable** α :

$$\mathit{eval} : \mathrm{term}\ \alpha \to \mathrm{value}\ \alpha$$

In that case, the type constructors "`term`" and "`value`" are **phantom types**. In our example, what is that piece of information exactly?

Encoding a predicate using a type

What is the meaning of "`e : term` α"?

Encoding a predicate using a type

What is the meaning of "`e : term` α"?
« The expression e has type α. »
In other words, we are encoding at the level of types the predicate that represents « well-typed-ness » in our typed expression language.
To instantiate this predicate, we also need to encode the syntax of the types for our expressions as types in the host programming language. This can be done using type constructors :

```
(* τ ::= int | τ × τ *)
type int_type
type (α, β) pair_type
```

A language of **well-typed** expressions

```
(* t ::= 0, 1, ... | π1 t | π2 t | (t, t) *)
type α term =
  | Lit : int → int_type term
  | Pair : α term × β term → (α, β) pair_type term
  | Fst : (α, β) pair_type term → α term
  | Snd : (α, β) pair_type term → β term

(* v ::= 0, 1, ... | (v, v) *)
type α value =
  | VInt : int → int_type value
  | VPair : α value × β value → (α, β) pair_type value
```

This piece of code declares two GADTs `term` and `value`.

An interpreter for well-typed expressions

```
let rec eval : type a. a term → a value = function
  | Lit x →
    (* a = int_ty *)
    VInt x
  | Pair (t1, t2) →
    (* ∃ b c.  a = (b, c) pair_ty, t1 :  b term, t2 :  c term *)
    VPair (eval t1, eval t2)
  | Fst t →
    (* ∃ b.  t :  ((a, b) pair_ty) term.  *)
    (match eval t with VPair (v1, _) → v1)
  | Snd t →
    (* ∃ b.  t :  ((b, a) pair_ty) term.  *)
    (match eval t with VPair (_, v2) → v2)
```

Type equivalence modulo type equalities

Following the usual ML typing rules, we know that

$$\texttt{VPair (eval t1, eval t2)} : \mathsf{value}\ (\beta \times \gamma)$$

This type is syntactically different from the expected type « value α ».

Fortunately, there is a local type equality in this branch which is « $\alpha = \beta \times \gamma$ ».

Thus, the work of the type-checker amounts to prove that:

$$\alpha = \beta \times \gamma \models \mathsf{value}\ \alpha = \mathsf{value}\ (\beta \times \gamma)$$

which is clearly true (given that ML types behave like first-order terms).

An interpreter for well-typed expressions

```
let rec eval : ... = function
  | Lit x → ...
  | Pair (t1, t2) → ...
  | Fst t →
      (* ∃γ, t : (pair_ty (α, γ))term *)
      begin match eval t with
      (* The only pattern of type (pair_ty (α, γ)) value. *)
      | VPair (v1, _) → v1
      end
  | Snd t → ...
```

Tagless interpreter

Since the previous program only uses **irrefutable** patterns in the inner pattern matchings, the **runtime tag attached to each value is useless**.

Thus, we can safely remove it by defining "`type` α `value =` α".

In other words, we can use the host programming language (equipped with GADTs) both as a language to write our interpreter in and as a language to **reflect the values and the types of our hosted expression language**.

Example : A language of **well-typed** expressions

Taking into account the new definition of `value`, we get a new declaration for the GADT `term`:

```
(* t ::= 0, 1, ... | π1 t | π2 t | (t, t) *)
type α term =
  | Lit : int → int term
  | Pair : α term × β term → (α × β) term
  | Fst : (α × β) term → α term
  | Snd : (α × β) term → β term

(* v ::= 0, 1, ... | (v, v) *)
type α value = α
```

Example : A tagless interpreter for well-typed expressions

```
let rec eval : type a. a term → a = function
  | Lit x →
    (* a = int_ty *)
    x
  | Pair (t1, t2) →
    (* ∃ b c.  a = (b * c), t1 :  b term, t2 :  c term *)
    (eval t1, eval t2)
  | Fst t →
    (* ∃ b.  t :  (a * b) term.  *)
    fst (eval t)
  | Snd t →
    (* ∃ b.  t :  (b * a) term.  *)
    snd (eval t)
```

Exercise

Exercise

Write a tagless interpreter for simply typed lambda calculus. □

λ-term with de Bruijn indices

The only complication comes from free variables. Indeed, a well-typed $\lambda-$term makes use of variables in compliance with the type ascribed to these variables when they were introduced in the context. If variables are represented by De Bruijn indices, typing environments Γ can be represented by a tuple of types.

```
type (Γ, α) term =
  | Var : (Γ, α) index → (Γ, α) term
  | App : (Γ, α → β) term × (Γ, α) term → (Γ, β) term
  | Lam : (α × Γ, β) term → (Γ, α → β) term

and (Γ, α) index =
  | Zero : (α × Γ, α) index
  | Shift : (Γ, α) index → (β × Γ, α) index
```

A tagless interpreter for simply typed λ-calculus

```
let rec lookup : type g a. (g, a) index → g → a = fun i env →
  match i with
  | Zero → fst env
  | Shift i → lookup i (snd env)

let rec eval : type g a. (g, a) term → g → a = fun t env →
  match t with
  | Var i → lookup i env
  | App (t1, t2) → (eval t1 env) (eval t2 env)
  | Lam t → fun x → eval t (x, env)
```

Digression

Can a tagless interpreter be implemented differently?

Yes, directly!

The idea is to implement each data constructors of the expression language directly by the code that implements its evaluation rule.

```
let varZ env = fst env
let varS vp env = vp (snd env)
let b (bv: bool) env = bv
let lam e env = fun x → e (x, env)
let app e1 e2 env = (e1 env) (e2 env)

let t = app (lam varZ) (b true)
```

This approach is deeply investigated in this paper [Carette et al., 2009].

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

- Embedding domain-specific languages
- Generic programming
- Dependently-typed programming
- Typed intermediate languages

GADTs in other type systems

Dynamic representation of types

Here is a GADT whose values are **dynamic representations** for types.

```
type (α, β) either =
      Left of α
    | Right of β

type α repr =
    | TyInt : int repr
    | TySum : α repr × β repr → ((α, β) either) repr
    | TyProd : α repr × β repr → (α × β) repr
```

Dynamic representation of types

The previous type can be used to implement generic type-directed functions :

```
let rec print : type a. a → a repr → string =
  fun x r →
    match r with
    | TyInt →
      (* "α = int", so "x : int". *)
      string_of_int x
    | TySum (ty1, ty2) →
      (* "∃βγ. α = β + γ", so "x : β + γ"*)
      (match x with
        | Left x1 → "L" ^ print x1 ty1
        | Right x2 → "R" ^ print x2 ty2)
    | TyProd (ty1, ty2) →
      (* "∃βγ. α = β * γ", so "x : β * γ"*)
      "(" ^ print (fst x) ty1 ^ ", "^ print (snd x) ty2 ^")"
```

Printf

Exercise

Program your own well-typed `printf` using the same idea. □

In fact, a well-typed `printf` can also be achieved, but with more efforts, in vanilla ML [Danvy, 1998].

Generic functions : Scrap your boilerplate (Reloaded)

A **generic** function is an overloaded function defined uniformly over the structure of data types through a generic view of these types. Indeed, a value of a data type is a tree of data constructor applications. Thus, the following type:

$$
\begin{array}{l}
\text{type } \alpha \text{ spine} = \\
\quad \mid \text{Constr} : \alpha \to \alpha \text{ spine} \\
\quad \mid \text{App} : (\alpha \to \beta) \text{ spine} \times \alpha \text{ with\_repr} \to \beta \text{ spine} \\
\\
\text{and } \alpha \text{ with\_repr} = \alpha \times \alpha \text{ repr}
\end{array}
$$

defines a **view** for any value of any data type.

Generic functions : Scrap your boilerplate (Reloaded)

Using the `repr` type we introduced earlier, a generic function can be defined over the structure of a type τ in order to convert any value of type τ into a value of type τ *spine* :

```
let rec to_spine : type a. a → a repr → a spine = fun x → function
  | TyInt → Constr x
  | TyUnit → Constr ()
  | TySum (ty1, ty2) →
    begin match x with
      | Left y → App (Constr mkLeft, (y, ty1))
      | Right y → App (Constr mkRight, (y, ty2))
    end
  | TyProd (ty1, ty2) →
    App (App (Constr mkPair, (fst x, ty1)), (snd x, ty2))

let rec from_spine : type a. a spine → a = function
  | Constr x → x
  | App (f, (x, _)) → (from_spine f) x
```

Generic functions : Scrap your boilerplate (Reloaded)

These definitions are sufficient to write generic functions:

```
let rec sum : type a. a → a repr → int = fun x → function
  | TyInt → x
  | r → sum_ (to_spine x r)

and sum_ : type a. a spine → int = function
  | Constr _ → 0
  | App (k, (x, ty)) → sum_ k + sum x ty
```

This kind of generic functions can be grouped under the class of queries:

```
type β query = { query : α. α → α repr → β }
```

Generic functions : Scrap your boilerplate (Reloaded)

Queries are naturally equipped with the following two combinators:

The combinator `map_query` broadcasts a query to the substructures.

```
let list_fold_left f init xs =
  let rec aux accu = function
    | [] → accu
    | x :: xs → f (aux accu xs) x
  in aux init xs

let rec reduce f = function
  | [] → assert false
  | x :: xs → list_fold_left f x xs

let rec map_query : type a. a query → (a list) query =
  fun q → { query = fun x r → map_query_ q (to_spine x r) }
```

Generic functions : Scrap your boilerplate (Reloaded)

The combinator `apply_query` combines the results of the queries on the toplevel element of the data and its substructures:

```
let rec apply_query : type b. (b → b → b) → b query → b query =
  fun f q →
    { query = fun x r →
      reduce f ((q.query x r)
                  :: (map_query (apply_query f q)).query x r)
    }
```

Generic functions : Scrap your boilerplate (Reloaded)

We eventually get a very succinct definition for the generic sum function:

```
let int_query : type a. a → a repr → int =
  fun x → function TyInt → x | r → 0
```

Generic functions : Scrap your boilerplate (Reloaded)

This GADT-based generic programming highly relies on the `repr` type.

Yet, there is neither a formalization nor an implementation of **extensible** GADTs at the moment. Therefore, the `repr` type is closed and it must be general enough to handle the entire type algebra. It is possible to devise a set of type operators that works as a basis for a large class of **representable** types [Yallop and Kiselyov, 2010].

Unfortunately, the resulting dynamic representation of types is not efficient: given a type definition « $\varepsilon\ \overline{\alpha} \equiv \tau$ », the size of the representation of $\varepsilon\ \overline{\tau}$ will be proportional to its definition τ.

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

- Embedding domain-specific languages
- Generic programming
- Dependently-typed programming
- Typed intermediate languages

GADTs in other type systems

Dependently-typed programming

In dependently-typed programming language, types are parametrized by values. This kind of types are useful to statically relate values to other values or to denote predicates over some values.

In its most general setting, the syntax of dependent types includes the syntax of expressions, which blurs the distinction between compile-time and run-time computations. This feature highly complicates the type-checking process.

GADTs offer a restricted form of dependent types that respects the phase distinction, that is a clear distinction between compile-time and run-time terms.

Singleton types

```
type zero
type α succ
type α nat =
  | Z : zero nat
  | S : α nat → (α succ) nat
```

- `Z` is the only value of type `zero nat`.
- `S Z` is the only value of type `zero succ nat`.
- The values of type `α succ nat` for some α are the unary representations of the positive natural numbers.

More generally, every type of the form « (zero succ*)nat » is a **singleton** type. As suggested by its name, a singleton type contains only one value.

How far can we go with that kind of dependent types?

How would we write the function that appends two lists?

First, we need a way to denote the addition of two natural numbers. Let us introduce a binary type constructor to represent it:

```
type (α, β) add
```

Second, write a suitable specification for `append`:

```
let rec append : type n m. n list → m list → ((n, m) add) vector =
function Nil → fun v → v
       | Cons (x, xs) → fun v → Cons (x, append xs v)
```

Is it well-typed?

How far can we go with that kind of dependent types?

```
let rec append
: type n m. n list → m list → ((n, m) add) vector =
function
| Nil →
   (* n = zero *)
   fun v → v
| Cons (x, xs) →
   (* ∃γ.n = γ succ *)
   fun v → Cons (x, append xs v)
```

This program is ill-typed because the type-checker failed to prove:

- $n = \text{zero} \models m = (n, m)\ \text{add}$
- $n = \gamma\ \text{succ} \models ((\gamma, m)\ \text{add})\ \text{succ} = (\gamma\ \text{succ}, m)\ \text{add}$

How far can we go with that kind of dependent types?

To fix that problem, the type-checker should be aware of the equational laws related to the natural numbers addition. There are several ways to do that:

- Generalize the syntax of constraints to authorize universal quantification, so that equational axioms of arithmetic can be written.
  This setting is too general: the entailment is undecidable.
- Allow the programmer to declare a set of toplevel universally quantified equalities that defines the type-level function `add`. These declarations must obey some rules to make sure that the entailment is still decidable.
  In Haskell, through the so-called **open type functions**.
- Extend the syntax of types to handle arbitrary type-level computation. In that case, the function `add` is defined using a lambda-term. The entailment decision is based on $\beta-$equivalence, which forces every type-level computation to always terminate.

Or ...

How far can we go with that kind of dependent types?

We can reformulate a bit our type definitions:

type (α, β, γ) add $=$
| AddZ : forall α. add (zero, α, α)
| AddS : forall α β. add $(\alpha, \beta, \gamma) \rightarrow$ add (succ α, β, succ γ)

The type constructor `add` is now ternary and it denotes an inductive relation between the two natural numbers and the result of their addition.

Values of GADTs as proof terms

The append function can be updated accordingly:

```
let rec append
: type n m o. (n, m, o) add → (α, n) list → (α, m) list → (α, o) list =
function
| AddZ →
   (* n = zero, m = o *)
   (function Nil → fun v → v)
| AddS p →
   (* ∃kl.n = k succ, m = l succ *)
   (function Cons (x, xs) → fun v → Cons (x, append p xs v))
```

This means that, in order to use that function, the programmer must **explicitly provide a proof-term**. This approach delegates the proof of the entailment relation to the programmer.

What kind of proof is it?

Proof-terms built using GADTs are essentially proofs of equality between types. There is a GADT that represents one equality proof between two types:

```
type (α, β) eq =
  | Refl : (α, α) eq
```

Theorem ([Johann and Ghani, 2008])

Every GADT can be reduced to one involving only the equality GADT `eq` *and existential quantification.*

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

- Embedding domain-specific languages
- Generic programming
- Dependently-typed programming
- Typed intermediate languages

GADTs in other type systems

A representation for type-class dictionaries

In the previous lecture, a remark highlighted a flexibility in the representation of type-class dictionaries. Indeed, the elaboration of a call « u » to an overloaded function of class C is translated into an **explicit dictionary passing** « $u\,q$ » where « q » is an expression that evaluates into the dictionary.

Dictionaries are usually represented as records. Another representation is based on a GADT. This GADT-based transformation is called **concretization** [Pottier and Gauthier, 2006].

Type-class dictionary typed by a GADT

The idea is to define, for each class C, a GADT whose data constructors are the representation of the instance declarations.

Type-class dictionary typed by a GADT

For instance, the following mini-Haskell declarations in the source language:

class Eq (X) { equal : X → X → bool }
inst Eq (Int) { equal = ... }
inst Eq (X) ⇒ Eq (list X) { equal = ... }

would lead to the following GADT declaration:

type α eq =
| EqInt : int eq
| EqList : α eq → α list eq

Where should the code for the class methods be put?

Interpretation of instance evidence

A value of type « eq τ » acts an evidence for the existence of an instance of the type-class« *Eq* » for a particular type « τ ».

This evidence must be interpreted to extract the exact definition of a particular method. For each method, we define an interpretation function to that mean.

```
let rec eq_int x y = (x = y)

and eq_list dict l1 l2 =
  match l1, l2 with
  | [], [] → true
  | x :: xs, y :: ys → equal dict x y
  | _ → false

and equal : eq α → α → α → bool = function
| EqInt → eq_int
| EqList dict → eq_list dict
```

Interpretation of instance evidence

In the same way, a declaration of a type-class that has a superclass:

```
class Eq (X) ⇒ Ord (X) { lt : X → X → Int }
```

. . . is translated into a GADT « α `ord` » as shown in the previous slide, plus a function that realizes the superclass relation of `Eq (X) ⇒ Ord (X)` :

```
let rec getEqFromOrd : type a. a ord → a eq = function
| OrdInt → EqInt
| OrdList d → EqList (getEqFromOrd d)
```

More powerful elaboration

The « non overlapping instance definitions » property can be rephrased in the translated GADT declarations: for each GADT C that reflects a class C, there cannot be two data constructors which instantiate C type parameter with the same type.

The following pattern matching is exhaustive for this reason:

```
let getEqFromEqList : type a. a list eq → a eq = function
| EqList dict → dict
```

This function realizes the rule « $\forall\alpha$.Eq (List α) $\rightarrow$ Eq α », which can be integrated in the proof search process of the elaboration. In the record-based representation, such a function cannot be written because the dictionary for « α `eq` » is hidden inside the environment of the closure that built the dictionary « α `list eq` ».

Contents

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

GADTs in other type systems

Let's look for an encoding of GADTs

Can we encode GADTs in other (rich) type systems?

Yes, using a Church-style encoding

A generalized algebraic data type is a special class of inductive data types. As a consequence, we can use a Church-style encoding to encode it into F_ω[Pfenning and Lee].

$$
\begin{array}{rcl}
t & ::= & x \mid \lambda x : \tau.t \mid t\ t \mid \Lambda\alpha :: K.t \mid t\ [\tau] \\
\tau & ::= & \alpha \mid \forall\alpha :: K.\tau \mid \lambda\alpha :: K.\tau \mid \tau \to \tau \mid \tau\ \tau \\
K & ::= & \star \mid K \Rightarrow K
\end{array}
$$

Recall that F_ω is an extension of system F that admits λ-abstraction at the type-level.

(We will come back on its definition in a next lecture.)

Church-style encoding of natural numbers

In an untyped setting, Church-style encoding of natural numbers is based on the following two definitions:

$$
\begin{array}{lcl}
\mathit{succ} & \equiv & \lambda x.\lambda s\ z.s\,(x\ s\ z) \\
\mathit{zero} & \equiv & \lambda s\ z.z
\end{array}
$$

What is the typed version of these data constructors (in System F)?

Church-style encoding of natural numbers

In a Church-style encoding, the representation of a data constructor K is a λ-term that implements the decomposition of K in an inductively defined computation.

Let us denote by X the type of this computation. Then :

$$
\begin{aligned}
\mathit{succ} &\equiv \lambda(x : \mathit{Nat}).\Lambda X.\lambda(s : X \to X)\ (z : X).s\ (x\ X\ s\ z) \\
\mathit{zero} &\equiv \Lambda X.\lambda(s : X \to X)\ (z : X).z \\
\text{with} & \\
\mathit{Nat} &\equiv \forall X.(X \to X) \to X \to X
\end{aligned}
$$

Type definition for the GADT type constructor "term"

$$
\begin{array}{rcl}
\mathsf{term}\ \alpha & \equiv & \forall \Theta : \star \Rightarrow \star. \\
\text{(for case Lit)} & & \quad (\mathsf{int} \to \Theta\ \mathsf{int}) \to \\
\text{(for case Pair)} & & \quad (\forall \beta\gamma.\ \Theta\ \beta \to \Theta\ \gamma \to \Theta\ (\beta \times \gamma)) \to \\
\text{(for case Fst)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \beta) \to \\
\text{(for case Snd)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \gamma) \to \\
 & & \Theta\ \alpha
\end{array}
$$

There are type quantifications under arrows: this is an F_2 term.

Type definition for the GADT type constructor "term"

$$
\begin{array}{rcl}
\mathsf{term}\ \alpha & \equiv & \forall \Theta : \star \Rightarrow \star. \\
\text{(for case Lit)} & & \quad (\mathsf{int} \to \Theta\ \mathsf{int}) \to \\
\text{(for case Pair)} & & \quad (\forall \beta\gamma.\ \Theta\ \beta \to \Theta\ \gamma \to \Theta\ (\beta \times \gamma)) \to \\
\text{(for case Fst)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \beta) \to \\
\text{(for case Snd)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \gamma) \to \\
 & & \quad \Theta\ \alpha
\end{array}
$$

What is Θ exactly?

Type definition for the GADT type constructor "term"

$$
\begin{array}{rcl}
\mathsf{term}\ \alpha & \equiv & \forall \Theta : \star \Rightarrow \star. \\
\text{(for case Lit)} & & \quad (\mathsf{int} \to \Theta\ \mathsf{int}) \to \\
\text{(for case Pair)} & & \quad (\forall \beta\gamma.\ \Theta\ \beta \to \Theta\ \gamma \to \Theta\ (\beta \times \gamma)) \to \\
\text{(for case Fst)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \beta) \to \\
\text{(for case Snd)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \gamma) \to \\
 & & \Theta\ \alpha
\end{array}
$$

What is Θ exactly?

It is a type function that denotes the typing context that has to be refined with respect to α.

Type definition for the GADT type constructor "term"

$$
\begin{array}{rcl}
\mathsf{term}\ \alpha & \equiv & \forall \Theta : \star \Rightarrow \star. \\
\text{(for case Lit)} & & \quad (\mathsf{int} \to \Theta\ \mathsf{int}) \to \\
\text{(for case Pair)} & & \quad (\forall \beta\gamma.\ \Theta\ \beta \to \Theta\ \gamma \to \Theta\ (\beta \times \gamma)) \to \\
\text{(for case Fst)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \beta) \to \\
\text{(for case Snd)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \gamma) \to \\
 & & \Theta\ \alpha
\end{array}
$$

How to read this type definition?

Type definition for the GADT type constructor "term"

$$
\begin{array}{rcl}
\text{term } \alpha & \equiv & \forall \Theta : \star \Rightarrow \star. \\
\text{(for case Lit)} & & \quad (\text{int} \to \Theta\ \text{int}) \to \\
\text{(for case Pair)} & & \quad (\forall \beta\gamma.\ \Theta\ \beta \to \Theta\ \gamma \to \Theta\ (\beta \times \gamma)) \to \\
\text{(for case Fst)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \beta) \to \\
\text{(for case Snd)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \gamma) \to \\
 & & \Theta\ \alpha
\end{array}
$$

How to read this type definition? For a type variable α and a typing context Θ that refers to α, a value v of type term α can be used to compute $\Theta\ \alpha$ if for each of the following cases :

(i) there exists $x : \text{int}$ and $\alpha = \text{int}$,

(ii) there exist $\beta, \gamma, c_1 : \Theta\ \beta, c_2 : \Theta\ \gamma$ and $\alpha = \beta \times \gamma$,

(iii) there exist $\beta, \gamma, c : \Theta\ (\beta \times \gamma)$ and $\alpha = \beta$,

(iv) there exist $\beta, \gamma, c : \Theta\ (\beta \times \gamma)$ and $\alpha = \gamma$;

it is provided a way to compute $\Theta\ \alpha$.

Type definition for the GADT type constructor "term"

$$
\begin{array}{rcl}
\mathsf{term}\ \alpha & \equiv & \forall \Theta : \star \Rightarrow \star. \\
\text{(for case Lit)} & & \quad (\mathsf{int} \to \Theta\ \mathsf{int}) \to \\
\text{(for case Pair)} & & \quad (\forall \beta\gamma.\ \Theta\ \beta \to \Theta\ \gamma \to \Theta\ (\beta \times \gamma)) \to \\
\text{(for case Fst)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \beta) \to \\
\text{(for case Snd)} & & \quad (\forall \beta\gamma.\ \Theta\ (\beta \times \gamma) \to \Theta\ \gamma) \to \\
 & & \quad \Theta\ \alpha
\end{array}
$$

The local type refinement is encoded by applying a substitution on "$\Theta\ \alpha$".

Term definition for term data constructor

$$
\begin{array}{rcl}
\mathsf{lit} & : & \mathsf{int} \to \mathsf{term\ int} \\
\mathsf{lit} & = & \lambda x : \mathsf{int}.\Lambda\Theta.\lambda \mathsf{lit\ pair\ fst\ snd}.\mathsf{lit}\ x \\
 & & \\
\mathsf{pair} & : & \forall\alpha\,\beta.\mathsf{term}\ \alpha \to \mathsf{term}\ \beta \to \mathsf{term}\ (\alpha \times \beta) \\
\mathsf{pair} & = & \Lambda\alpha.\Lambda\beta.\lambda t : \mathsf{term}\ \alpha.\lambda u : \mathsf{term}\ \beta. \\
 & & \quad \Lambda\Theta.\lambda \mathsf{lit\ pair\ fst\ snd}. \\
 & & \quad\quad \mathsf{pair}\ \alpha\ \beta \\
 & & \quad\quad\quad (t\ \Theta\ \mathsf{lit\ pair\ fst\ snd}) \\
 & & \quad\quad\quad (u\ \Theta\ \mathsf{lit\ pair\ fst\ snd})
\end{array}
$$

These terms have third-order polymorphic types. They select the right cases between the provided ones. They also take the responsibility of calling the interpretation of their sub-components.

Exercise: Write the F_3 term for fst.

Tagless interpreter in F_ω

$$
\begin{array}{lcl}
\mathsf{eval} & : & \forall\alpha.\mathsf{term}\ \alpha \to \alpha \\
\mathsf{eval} & = & \Lambda\alpha.\lambda p : \mathsf{term}\ \alpha. \\
 & & \quad p\ \lambda\alpha.\alpha \\
 & & \qquad (\Lambda\alpha.\lambda x.x) \\
 & & \qquad (\Lambda\alpha\beta.\lambda x\ y.(x,y)) \\
 & & \qquad (\Lambda\alpha\beta.\lambda(x,y).x) \\
 & & \qquad (\Lambda\alpha\beta.\lambda(x,y).y)
\end{array}
$$

To accept this term, the type-checker must **reduce** $(\lambda\alpha.\alpha)\ \alpha$ into α, which explain the need for F_ω in the general case. Yet, a language with higher-order kind polymorphism is sufficient in practice.

Scott-style encoding

There exist another, less spread than Church's, encoding for algebraic data types.

In Scott-style encodings, the term that encodes a data constructor does not take the responsibility of calling the interpretation of its sub-components.

For instance, the successor is now encoded by:

$$S \equiv \lambda x.\lambda s\, z.s\ x$$

This encoding has been shown to be more practical to encode GADT in a call-by-value setting [Mandelbaum and Stump].

Thus, is a primitive notion of GADT really relevant?

The weakness of these encodings lies in the necessity to **make explicit** the context into which the type refinement takes place.

For instance, our second example turns into :

$$
\begin{array}{rcl}
\mathsf{repr}\ \alpha & \equiv & \forall \Theta : \star \Rightarrow \star. \\
\text{(for case TyInt)} & & \quad (\Theta\ \mathsf{int}) \to \\
\text{(for case TySum)} & & \quad (\forall \beta\gamma.\ \Theta\ \beta \to \Theta\ \gamma \to \Theta\ (\beta + \gamma)) \to \\
\text{(for case TyProd)} & & \quad (\forall \beta\gamma.\ \Theta\ \beta \to \Theta\ \gamma \to \Theta\ (\beta \times \gamma)) \to \\
 & & \Theta\ \alpha
\end{array}
$$

Thus, is a primitive notion of GADT really relevant?

$$
\begin{array}{lcl}
\mathsf{print} & : & \forall\alpha.\alpha \to \mathsf{repr}\ \alpha \to \mathsf{string} \\
\mathsf{print} & = & \Lambda\alpha.\lambda(x:\alpha).\lambda(r:\mathit{repr}\ \alpha). \\
 & & \quad r\ ? \\
 & & \qquad (\mathsf{string\_of\_int}\ ?) \\
 & & \qquad (\Lambda\beta.\Lambda\gamma.\lambda(c_1:\Theta\ \beta).\lambda(c_2:\Theta\ \gamma).?) \\
 & & \qquad (\Lambda\beta.\Lambda\gamma.\lambda(c_1:\Theta\ \beta).\lambda(c_2:\Theta\ \gamma).?)
\end{array}
$$

Thus, is a primitive notion of GADT really relevant?

$$
\begin{array}{lcl}
\text{print} & : & \forall\alpha.\alpha \to \text{repr}\ \alpha \to \text{string} \\
\text{print} & = & \Lambda\alpha.\lambda(x : \alpha).\lambda(r : \mathit{repr}\ \alpha). \\
 & & \quad r\ \lambda\alpha.\text{string} \\
 & & \quad\quad (\text{string\_of\_int}\ ?) \\
 & & \quad\quad (\Lambda\beta.\Lambda\gamma.\lambda(c_1 : \text{string}.\lambda(c_2 : \Theta\ \gamma).?) \\
 & & \quad\quad (\Lambda\beta.\Lambda\gamma.\lambda(c_1 : \text{string}.\lambda(c_2 : \Theta\ \gamma).?)
\end{array}
$$

Thus, is a primitive notion of GADT really relevant?

$$
\begin{array}{lcl}
\mathsf{print} & : & \forall\alpha.\alpha \to \mathsf{repr}\ \alpha \to \mathsf{string} \\
\mathsf{print} & = & \Lambda\alpha.\lambda(x:\alpha).\lambda(r:\mathit{repr}\ \alpha). \\
 & & \quad (r\ \lambda\alpha.\alpha \to \mathsf{string} \\
 & & \quad\quad (\mathsf{string\_of\_int}\ ) \\
 & & \quad\quad (\Lambda\beta.\Lambda\gamma.\lambda(c_1:\beta \to \mathsf{string}).\lambda(c_2:\gamma \to \mathsf{string}). \\
 & & \quad\quad\quad \lambda(x':\beta+\gamma)....) \\
 & & \quad\quad (\Lambda\beta.\Lambda\gamma.\lambda(c_1:\beta \to \mathsf{string}).\lambda(c_2:\gamma \to \mathsf{string}). \\
 & & \quad\quad\quad \lambda(x':\beta\times\gamma)....))\ x
\end{array}
$$

Thus, is a primitive notion of GADT really relevant?

$$
\begin{array}{lcl}
\text{print} & : & \forall\alpha.\alpha \to \text{repr } \alpha \to \text{string} \\
\text{print} & = & \Lambda\alpha.\lambda(x:\alpha).\lambda(r:\mathit{repr}\ \alpha). \\
 & & \quad (r\ \lambda\alpha.\alpha \to \text{string}\ldots)\ x
\end{array}
$$

An extra term-level redex has been introduced to have this program accepted by the type-checker of F_ω.

Are GADTs in Coq?

The Calculus of Inductive Construction offers general inductive types. Yet, the refinement of inductive types through dependent inversion is not applied to the entire typing environment because this would break subject reduction. As a consequence, in Coq, the typing context to be refined must also be explicitly ascribed in the return clause of pattern matchings :

```
Fixpoint print (A : Type) (x : A) (r : repr A) : string :=
  (match r in repr T return (T → string) with
    | TyNat ⇒ fun (x : nat) ⇒ string_of_nat x
    | TySum B C ty1 ty2 ⇒ fun (x : B + C) ⇒
      match x with
        | inl x1 ⇒ print B x1 ty1
        | inr x2 ⇒ print C x2 ty2
      end
    | TyProd B C ty1 ty2 ⇒ fun (x : B × C) ⇒
      append (print B (fst x) ty1) (print C (snd x) ty2)
  end) x.
```

Bibliography I

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796809007205.

James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

Olivier Danvy. Functional unparsing. *J. Funct. Program.*, 8:621–625, November 1998. ISSN 0956-7968. doi: 10.1017/S0956796898003104.

Patricia Johann and Neil Ghani. Foundations for structured programming with gadts. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 297–308, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: http://doi.acm.org/10.1145/1328438.1328475.

Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California*, pages 78–91, June 1992.

Bibliography II

- Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL '99, pages 109–122, New York, NY, USA, 1999. ACM. ISBN 1-58113-255-7. doi: http://doi.acm.org/10.1145/331960.331977.
- Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, New York, NY, USA, 1974. ACM. doi: http://doi.acm.org/10.1145/800233.807045.

Yitzhak Mandelbaum and Aaron Stump. Gadts for the ocaml masses. Technical report.

Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322186.322198.

Frank Pfenning and Peter Lee. Leap: A language with eval and polymorphism.

- François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.

Bibliography III

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

Jeremy Yallop and Oleg Kiselyov. First-class modules: hidden power and tantalizing promises. ACM SIGPLAN Workshop on ML, September 2010. Baltimore, Maryland, United States.