

Rich types, Tractable typing – Dependently-typed programming languages –

Yann Régis-Gianas
yrg@irif.fr

2018-12-21

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

GADTs encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

GADTs through a motivating example

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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

Algebraic data types : data constructors

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Consider an OCaml type definition for list of integers:

```
1  type ilist =  
2    | Nil  
3    | 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.

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Algebraic data types : pattern matching

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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:

```
1  let hd l =  
2    match l with  
3    | Nil -> (* The list "l" is empty. *) failwith "hd"  
4    | Cons (x, _) -> (* The list "l" is not empty. *) x
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

A safer head extraction?

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax
Operational semantics
Type System
Metatheory
Implementation of a type checker
for MLG1

Applications

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

GADTs encodings

Bibliography

A trick in vanilla ML : Phantom types [1]

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

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

GADTs encodings

Bibliography

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 'a of a value v of phantom type $\text{'a} \rightarrow \text{t}$ will not represent the type of a component of v but a static property attached to v .

A phantom type for emptiness

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

```
1  type 'a plist = ilist
```

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

```
1  type empty  
2  type nonempty
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Smart constructors to define a phantom type

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

By turning this type into an abstract data type [2], we can specialize the type of the data constructors for lists:

```
1 module List : sig
2   type 'a plist
3   val nil : empty plist
4   val cons : int -> 'a plist -> nonempty plist
5   val hd : nonempty plist -> int
6 end = struct
7   type ilist = Nil | Cons of int * ilist
8   type 'a plist = ilist
9   let nil = Nil
10  let cons x xs = Cons (x, xs)
11  let hd = function Cons (x, _) -> x | Nil -> assert false
12 end
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

A safer (?) head extraction

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

```
1  let hd : nonempty List.plist -> int = function
2    | Nil -> assert false
3    | 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.

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax
Operational semantics
Type System
Metatheory
Implementation of a type checker
for MLG1

Applications

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

GADTs encodings

Bibliography

Limit of phantom types

```
1  let totalHd l =  
2    if l = nil then 42 else 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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Generalized Algebraic Data Types

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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

```
1  (* This code fragment is accepted by ocaml >= 4.00 *)
2  type 'a glist =
3      | Nil : empty glist
4      | Cons : int * 'a 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.**

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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`:

```
1  let hd : nonempty glist -> int = function
2    | 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`.

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Automatic type refinement in action

Besides, a type-checker for GADTs is able to **refine the type of 1** in the second case of this pattern matching:

```
1  let totalHd : type a. a glist -> int = function
2    | Nil -> 42
3    | (Cons _ ) as l -> hd l
```

In the body of the last branch, the type-checker knows that the **term 1 has both the type a glist and the type nonempty glist** because **1** has been built by application of the data constructor **Cons**. The second type is enough to accept the application of **hd** on **1**.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Where does the magic lie?

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax
Operational semantics
Type System
Metatheory
Implementation of a type checker
for MLG1

Applications

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

GADTs encodings

Bibliography

Type equalities

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

```
1  type 'a glist =  
2    | Nil : ['a = empty] 'a glist  
3    | Cons : ['a = nonempty] int * 'b glist -> 'a glist
```

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

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Type-checking totalHd

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas

yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

```
1  let totalHd : type a. a glist -> int = function
2    | Nil -> 42
3    | (Cons _) as l -> hd l
4      (* Here, a = nonempty. *)
5      (* As  $\Gamma \vdash l : a \text{ glist}$  *)
6      (* and a = nonempty  $\vdash a \text{ glist} = \text{nonempty glist}$ , *)
7      (* by conversion,  $\Gamma \vdash l : \text{nonempty glist}$  *)
```

Origins of GADTs

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Generalized Algebraic Data types have been first introduced independently by Hongwei Xi, Chiyang Chen and Gang Chen [3] and by James Cheney and Ralf Hinze [4].

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.

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Exercise

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Consider the following two type definitions:

```
1  type zero
2  type 'a succ = S of 'a
```

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.

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Exercises

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

② Consider the following recursive function:

```
1  let rec sum l1 l2 =  
2    match l1, l2 with  
3    | [], [] -> []  
4    | x :: xs, y :: ys -> (x + y) :: sum xs ys  
5    | _ -> failwith ``sum``
```

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

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Exercises

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

③ Examine the following function:

```
1  let rec append l1 l2 =  
2    match l1 with  
3    | [] -> l2  
4    | x :: xs -> x :: (append xs l2)
```

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

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

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 encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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.

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

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

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 encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Syntax of MLGI

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Terms	t	$::=$
Variable		x
Function		$\lambda x.t$
Function application		$t\ t$
Local definition		$\text{let } x = t \text{ in } t$
Fixpoint		$\mu x.t$
Data constructor application		$K(t, \dots, t)$
Case analysis		$\text{match } t \text{ with } \bar{c}$

Syntax of MLGI (continued)

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

Clauses $c ::= p \Rightarrow t$

Patterns $p ::= K(\bar{x})$

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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

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 encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Values

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

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

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Evaluation context

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Evaluation context	$\mathbb{E} ::=$
Hole	\square
LHS of an application	$ \mathbb{E} \ t$
RHS of an application	$ v \ \mathbb{E}$
Left hand side of a let	$ \text{let } x = \mathbb{E} \text{ in } t$
Right hand side of a let	$ \text{let } x = v \text{ in } \mathbb{E}$
Arguments of constructors	$ K(v_1, \dots, v_n, \mathbb{E}, t_1, \dots, t_k)$
Scrutinee evaluation	$ \text{match } \mathbb{E} \text{ with } \bar{c}$

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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

Reduction rules

$$\begin{array}{ll} \text{(E-Context)} & \mathbb{E} [t] \rightarrow \mathbb{E} [t'] \\ & \text{if } t \rightarrow t' \\ \text{(E-Beta)} & (\lambda x.t)v \rightarrow [x \mapsto v]t \\ \text{(E-Let)} & \text{let } x = v \text{ in } t \rightarrow [x \mapsto v]t \\ \text{(E-Fix)} & \mu x.t \rightarrow [x \mapsto \mu x.t]t \end{array}$$

$$\begin{array}{ll} \text{(E-Match)} & \text{match } K(v_1, \dots, v_n) \text{ with } K(x_1, \dots, x_n) \Rightarrow t \mid \bar{c} \\ & \rightarrow [x_1 \mapsto v_1] \dots [x_n \mapsto v_n]t \end{array}$$

$$\begin{array}{ll} \text{(E-NoMatch)} & \text{match } K'(v_1, \dots, v_m) \text{ with } K(x_1, \dots, x_n) \Rightarrow t \mid \bar{c} \\ & \rightarrow \text{match } K'(v_1, \dots, v_m) \text{ with } \bar{c} \\ & \text{if } K \neq K' \end{array}$$

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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** $K(v_1, \dots, v_m)$ **with** \emptyset ”. Hence, a type-checker must ensure that pattern matchings are exhaustive.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

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 encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Syntax for types

Types	$\tau ::=$
Type variable	α
Function type	$ \tau \rightarrow \tau$
Type application	$ \varepsilon \overline{\tau \tau}$
Type schemes	$\sigma ::= \forall \overline{\alpha}. \tau$

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

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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:

$$K :: \forall \bar{\beta} \bar{\alpha}. \tau_1 \times \dots \times \tau_m \rightarrow \varepsilon \bar{\alpha} \bar{\tau}$$

with $\bar{\beta} \# \bar{\alpha}, \bar{\beta} \in \text{FTV}(\bar{\tau})$ and $\bar{\alpha}$ are all distinct.

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

$$\forall \bar{\beta} \bar{\alpha}. \tau_1 \times \dots \times \tau_m \rightarrow \varepsilon \bar{\alpha} \bar{\tau} \preceq \sigma$$

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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:

$$K :: \forall \overline{\alpha}. \tau_1 \times \dots \times \tau_m \rightarrow \varepsilon \overline{\alpha}$$

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:

$$\begin{aligned} \text{Nil} &:: \forall \alpha. \text{list } \alpha \\ \text{Cons} &:: \forall \alpha. \alpha \times \text{list } \alpha \rightarrow \text{list } \alpha \end{aligned}$$

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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 [5].

Example

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

Closure $:: \forall \beta \alpha_1 \alpha_2. \beta \times (\beta \rightarrow \alpha_1 \rightarrow \alpha_2) \rightarrow \text{closure } \alpha_1 \alpha_2$

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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:

$$\begin{aligned}\text{Nil} &:: \forall \alpha. \text{list } \alpha \text{ zero} \\ \text{Cons} &:: \forall \beta \alpha. \alpha \times \text{list } \alpha \beta \rightarrow \text{list } \alpha (\text{succ } \beta)\end{aligned}$$

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Syntax for type equalities

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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:

Equation systems $E ::= \mathbf{true} \mid \tau = \tau \mid E \wedge E$

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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 .”

“ $p : \tau \vdash (\bar{\beta}, E, \Gamma)$ ”, read

“The assumption ‘the pattern p has type τ ’ introduces local variables $\bar{\beta}$, local equations E and a local environment Γ .”

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Well-typed terms

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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 “ $E \models \text{false}$ ”, if there is no such ϕ .

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Rule for data constructor applications

CSTR

$$\frac{K \preceq \tau_1 \times \dots \times \tau_n \rightarrow \varepsilon \bar{\tau}_1 \bar{\tau}_2 \quad \forall i \quad E, \Gamma \vdash t_i : \tau_i}{E, \Gamma \vdash K(t_1 \dots t_n) : \varepsilon \bar{\tau}_1 \bar{\tau}_2}$$

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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Conversion rule

$$\text{CONV} \quad \frac{E, \Gamma \vdash t : \tau_1 \quad E \models \tau_1 = \tau_2}{E, \Gamma \vdash t : \tau_2}$$

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:

$$\begin{array}{c} \frac{}{E_1 \wedge \tau_1 = \tau_2 \wedge E_2 \models \tau_1 = \tau_2} \\[1em] \frac{E \models \bar{\tau}_1 = \bar{\tau}'_1 \quad E \models \bar{\tau}_2 = \bar{\tau}'_2}{E \models \varepsilon \bar{\tau}_1 \bar{\tau}_2 = \varepsilon \bar{\tau}'_1 \bar{\tau}'_2} \\[1em] \frac{E \models \tau_1 = \tau_2 \quad E \models \tau_2 = \tau_3}{E \models \tau_1 = \tau_3} \end{array} \quad \frac{E \models \tau_1 = \tau'_1 \quad E \models \tau_2 = \tau'_2}{E \models \tau_1 \rightarrow \tau_2 = \tau'_1 \rightarrow \tau'_2}$$

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Rule for pattern matchings

CASE

$$\frac{E, \Gamma \vdash t : \tau_1 \quad \forall i \quad E, \Gamma \vdash c_i : \tau_1 \rightarrow \tau_2}{E, \Gamma \vdash \text{match } t \text{ with } c_1 \dots c_n : \tau_2}$$

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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Rule for clauses

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

CLAUSE

$$\frac{\begin{array}{c} p : \varepsilon\bar{\tau}_1\bar{\tau}_2 \vdash (\bar{\beta}, E', \Gamma') \\ \bar{\beta} \# \text{FTV}(E, \Gamma, \tau_2) \quad E \wedge E', \Gamma\Gamma' \vdash t : \tau_2 \end{array}}{E, \Gamma \vdash p \Rightarrow t : \varepsilon\bar{\tau}_1\bar{\tau}_2 \rightarrow \tau_2}$$

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

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Extraction of type equalities

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

PAT

$$\frac{K \preceq \forall \bar{\beta}. \tau_1 \times \dots \times \tau_n \rightarrow \varepsilon \bar{\tau}_1 \bar{\tau} \quad \bar{\beta} \# \text{FTV}(\bar{\tau}_1, \bar{\tau}_2)}{K (x_1 \dots x_n) : \varepsilon \bar{\tau}_1 \bar{\tau}_2 \vdash (\bar{\beta}, \bar{\tau}_2 = \bar{\tau}, (x_1 : \tau_1; \dots; x_n : \tau_n))}$$

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:

PAT-REGULAR

$$\frac{K \preceq \tau_1 \times \dots \times \tau_n \rightarrow \varepsilon \overline{\tau}_1}{K (x_1 \dots x_n) : \varepsilon \overline{\tau}_1 \vdash (\emptyset, \mathbf{true}, (x_1 : \tau_1; \dots; x_n : \tau_n))}$$

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.

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Specialization of Pat to iso-existential ADT

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

$$\frac{\mathsf{PAT} \quad K \preceq \forall \bar{\beta}. \tau_1 \times \dots \times \tau_n \rightarrow \varepsilon \bar{\tau}_1 \quad \bar{\beta} \# \mathsf{FTV}(\bar{\tau}_1)}{K \ (x_1 \dots x_n) : \varepsilon \bar{\tau}_1 \vdash (\bar{\beta}, \mathbf{true}, (x_1 : \tau_1; \dots; x_n : \tau_n))}$$

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Back to the general version

PAT

$$\frac{K \preceq \forall \bar{\beta}. \tau_1 \times \dots \times \tau_n \rightarrow \varepsilon \bar{\tau}_1 \bar{\tau} \quad \bar{\beta} \# \text{FTV}(\bar{\tau}_1, \bar{\tau}_2)}{K (x_1 \dots x_n) : \varepsilon \bar{\tau}_1 \bar{\tau}_2 \vdash (\bar{\beta}, \bar{\tau}_2 = \bar{\tau}, (x_1 : \tau_1; \dots; x_n : \tau_n))}$$

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

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

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

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 encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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.

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Key lemma

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

Lemma

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

- (i) $E'' \models E'$
- (ii) $\exists \bar{\beta}. E \equiv E''$
- (iii) $\forall i, E'', \cdot \vdash v_i : \tau_i$

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax
Operational semantics
Type System

Metatheory
Implementation of a type checker
for MLG1

Applications

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

GADTs encodings

Bibliography

Subject reduction

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Theorem (Subject reduction)

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

Proof.

By induction on the derivation of $t \rightarrow t'$.



Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Progress

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

match v 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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Improving progress thanks to dead code elimination

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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.

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Improving progress thanks to dead code elimination

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Let us complete every non exhaustive pattern matching with a branch « $p \Rightarrow \text{wrong}$ » 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**.

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Improving progress thanks to dead code elimination

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

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

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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

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 encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Type-checking algorithm

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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 admits 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.

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

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

An algorithm based on congruence closure

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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 [Nelson80].

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

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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

An algorithm based on eager application of MGU

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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.

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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

Plan

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 encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

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

GADTs encodings

Bibliography

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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

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 encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax
Operational semantics
Type System
Metatheory
Implementation of a type checker
for MLG1

Applications

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

GADTs encodings

Bibliography

Example : A language of expressions

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

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

```
1  (* t ::= 0, 1, ... |  $\pi_1 t$  |  $\pi_2 t$  | (t, t) *)
2  type term =
3    | Lit of int
4    | Pair of term * term
5    | Fst of term
6    | Snd of term
7
8  (* v ::= 0, 1, ... | (v, v) *)
9  type value =
10   | VInt of int
11   | VPair of value * value
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

A running example

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

An interpreter of this language is a straightforward inductive function:

```
1  let rec eval : term -> value = function
2    | Lit x -> VInt x
3    | Pair (t1, t2) -> VPair (eval t1, eval t2)
4    | Fst t -> begin match eval t with
5                  | VPair (v1, _) -> v1
6                  | _ -> failwith "Only pairs can be projected."
7                end
8    | Snd t -> ...
```

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

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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:

```
1  (* [eval e] interprets a well-typed expression of type T *)
2  (* into a value of type T, or more formally: *)
3  (*  $\forall eT, \text{Pre: } \vdash e : T'' \Rightarrow \text{Post: } \vdash \text{eval } e : T''$  *)
4  let rec eval : term -> value = function
5  | Lit x -> VInt x
6    (* By inversion of Pre, we have  $\vdash \text{Lit } x : \text{int}$  *)
7    (* Thus,  $T = \text{int}$  and, we indeed have,  $\vdash \text{VInt } x : \text{int}$  *)
8  | Pair (t1, t2) -> VPair (eval t1, eval t2)
9    (* By inversion of Pre, there exist  $\beta_1, \beta_2$  such that *)
10   (*  $\vdash \text{Pair } (t1, t2) : \beta_1 \times \beta_2, \vdash t1 : \beta_1$ , and  $\vdash t2 : \beta_2$ . *)
11   (* Thus,  $\vdash \text{eval } t1 : \beta_1$ , and  $\vdash \text{eval } t2 : \beta_2$ . *)
12   (* Eventually, we obtain  $\vdash (\text{eval } t1, \text{eval } t2) : \beta_1 \times \beta_2$ . *)
13  | Fst t -> ...
14  | Snd t -> ...
```

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Example : A commented interpreter for expressions

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

```
1  (* [eval e] interprets a well-typed expression of type T *)
2  (* into a value of type T, or more formally: *)
3  (*  $\forall eT, \text{Pre} : \vdash e : T' \Rightarrow \text{Post} : \vdash \text{eval } e : T'$  *)
4  let rec eval : term -> value = function
5  | ...
6  | Fst t -> begin match eval t with
7                | VPair (v1, _) -> v1
8                | _ -> failwith "Only pairs can be projected."
9            end
10     (* By inversion of Pre,  $\exists \beta, \vdash t : T \times \beta$  *)
11     (* Then,  $\vdash \text{eval } t : T \times \beta$  *)
12     (* And by the classification lemma,  $\text{eval } t = (v_1, v_2)$  *)
13     (* with  $\vdash v_1 : T$ , which is what we expected. *)
14 | Snd t -> ...
```

Example : A commented interpreter for expressions

```
1  let rec eval : term -> value = function ...
2    | Fst t -> begin match eval t with
3        | VPair (v1, _) -> v1
4        | _ -> assert false
5    end
6    (* Thus, by the classification lemma,  $\text{eval } t = (v_1, v_2)$  *)
7    | 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**.

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

How to encode a predicate using a type?

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

$$eval : \text{term } \alpha \rightarrow \text{value } \alpha$$

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

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Encoding a predicate using a type

What is the meaning of “`e : term 'a`”?

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Encoding a predicate using a type

What is the meaning of “`e : term 'a`”?

« 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 :

```
1  (*  $\tau ::= \text{int} \mid \tau \times \tau$  *)
2  type int_type
3  type ('a, 'b) pair_type
```

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax
Operational semantics
Type System
Metatheory
Implementation of a type checker
for MLG1

Applications

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

GADTs encodings

Bibliography

A language of **well-typed** expressions

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

```
1  (* t ::= 0, 1, ... |  $\pi_1 t$  |  $\pi_2 t$  | (t, t) *)
2  type 'a term =
3    | Lit : int -> int_type term
4    | Pair : 'a term * 'b term -> ('a, 'b) pair_type term
5    | Fst : ('a, 'b) pair_type term -> 'a term
6    | Snd : ('a, 'b) pair_type term -> 'b term
7
8  (* v ::= 0, 1, ... | (v, v) *)
9  type 'a value =
10    | VInt : int -> int_type value
11    | VPair : 'a value * 'b value -> ('a, 'b) pair_type value
```

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

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

An interpreter for well-typed expressions

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax
Operational semantics
Type System
Metatheory
Implementation of a type checker
for MLG1

Applications

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

GADTs encodings

Bibliography

```
1  let rec eval : type a. a term -> a value = function
2  | Lit x ->
3    (* a = int_ty *)
4    VInt x
5  | Pair (t1, t2) ->
6    (*  $\exists b\ c. a = (b, c)$  pair_ty, t1 : b term, t2 : c term *)
7    VPair (eval t1, eval t2)
8  | Fst t ->
9    (*  $\exists b. t : ((a, b)$  pair_ty) term. *)
10   (match eval t with VPair (v1, _) -> v1)
11 | Snd t ->
12   (*  $\exists b. t : ((b, a)$  pair_ty) term. *)
13   (match eval t with VPair (_, v2) -> v2)
```

Type equivalence modulo type equalities

Following the usual ML typing rules, we know that

VPair (**eval** t_1 , **eval** t_2) : 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:

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

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

An interpreter for well-typed expressions

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

```
1  let rec eval : ... = function
2    | Lit x -> ...
3    | Pair (t1, t2) -> ...
4    | Fst t ->
5      (*  $\exists \gamma, t : (pair\_ty(\alpha, \gamma)) term$  *)
6      begin match eval t with
7      (* The only pattern of type  $(pair\_ty(\alpha, \gamma)) value.$  *)
8      | VPair (v1, _) -> v1
9      end
10   | Snd t -> ...
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Tagless interpreter

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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** 'a **value** = 'a”.

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**.

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

```
1  (* t ::= 0, 1, ... |  $\pi_1 t$  |  $\pi_2 t$  | (t, t) *)
2  type 'a term =
3    | Lit : int -> int term
4    | Pair : 'a term * 'b term -> ('a * 'b) term
5    | Fst : ('a * 'b) term -> 'a term
6    | Snd : ('a * 'b) term -> 'b term
7
8  (* v ::= 0, 1, ... | (v, v) *)
9  type 'a value = 'a
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Example : A tagless interpreter for well-typed expressions

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

```
1  let rec eval : type a. a term -> a = function
2    | Lit x ->
3      (* a = int_ty *)
4      x
5    | Pair (t1, t2) ->
6      (*  $\exists b\ c. a = (b * c), t1 : b\ term, t2 : c\ term$  *)
7      (eval t1, eval t2)
8    | Fst t ->
9      (*  $\exists b. t : (a * b)\ term.$  *)
10     fst (eval t)
11    | Snd t ->
12     (*  $\exists b. t : (b * a)\ term.$  *)
13     snd (eval t)
```


Exercise

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Exercise 1:

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 λ -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.

```
1 type ('g, 'a) term =  
2   | Var : ('g, 'a) index -> ('g, 'a) term  
3   | App : ('g, 'a -> 'b) term * ('g, 'a) term -> ('g, 'b) term  
4   | Lam : ('a * 'g, 'b) term -> ('g, 'a -> 'b) term  
5  
6 and ('g, 'a) index =  
7   | Zero : ('a * 'g, 'a) index  
8   | Shift : ('g, 'a) index -> ('b * 'g, 'a) index
```

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

A tagless interpreter for simply typed λ -calculus

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

```
1 let rec lookup
2   : type g a. (g, a) index -> g -> a = fun i env ->
3     match i with
4     | Zero -> fst env
5     | Shift i -> lookup i (snd env)
6
7 let rec eval
8   : type g a. (g, a) term -> g -> a = fun t env ->
9     match t with
10    | Var i -> lookup i env
11    | App (t1, t2) -> (eval t1 env) (eval t2 env)
12    | Lam t -> fun x -> eval t (x, env)
```

Can a tagless interpreter be implemented
differently?

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Yes, directly!

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

```
1  let varZ env = fst env
2  let varS vp env = vp (snd env)
3  let b (bv:bool) env = bv
4  let lam e env = fun x -> e (x, env)
5  let app e1 e2 env = (e1 env) (e2 env)
6
7  let t = app (lam varZ) (b true)
```

This approach is deeply investigated in this paper [6].

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

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 encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax
Operational semantics
Type System
Metatheory
Implementation of a type checker
for MLG1

Applications

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

GADTs encodings

Bibliography

Dynamic representation of types

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

```
1  type ('a, 'b) either =  
2      Left of 'a  
3      | Right of 'b  
4  
5  type 'a repr =  
6      | TyInt   : int repr  
7      | TySum   : 'a repr * 'b repr -> (('a, 'b) either) repr  
8      | TyProd  : 'a repr * 'b repr -> ('a * 'b) repr
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Dynamic representation of types

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

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

```
1 let rec print : type a. a -> a repr -> string =  
2   fun x r ->  
3     match r with  
4     | TyInt ->  
5       (* `` $\alpha = \text{int}$ '', so `` $x : \text{int}$ '' . *)  
6       string_of_int x  
7     | TySum (ty1, ty2) ->  
8       (* `` $\exists \beta \gamma. \alpha = \beta + \gamma$ '', so `` $x : \beta + \gamma$ '' *)  
9       (match x with  
10        | Left x1 -> "L" ^ print x1 ty1  
11         | Right x2 -> "R" ^ print x2 ty2)  
12    | TyProd (ty1, ty2) ->  
13      (* `` $\exists \beta \gamma. \alpha = \beta * \gamma$ '', so `` $x : \beta * \gamma$ '' *)  
14      "(" ^ print (fst x) ty1 ^ ", " ^ print (snd x) ty2 ^ ")"
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Exercise 2:

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 [7].

Generic functions : Scrap your boilerplate (Reloaded)

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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:

```
1  type 'a spine =  
2    | Constr : 'a -> 'a spine  
3    | App   : ('a -> 'b) spine * 'a with_repr -> 'b spine  
4  
5  and 'a with_repr = 'a * 'a repr
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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* :

```
1  let rec to_spine : type a. a -> a repr -> a spine
2  = fun x -> function
3    | TyInt -> Constr x
4    | TyUnit -> Constr ()
5    | TySum (ty1, ty2) ->
6      begin match x with
7        | Left y -> App (Constr mkLeft, (y, ty1))
8        | Right y -> App (Constr mkRight, (y, ty2))
9      end
10   | TyProd (ty1, ty2) ->
11     App (App (Constr mkPair, (fst x, ty1)), (snd x, ty2))
12
13  let rec from_spine : type a. a spine -> a = function
14    | Constr x -> x
15    | App (f, (x, _)) -> (from_spine f) x
```

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Generic functions : Scrap your boilerplate (Reloaded)

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

These definitions are sufficient to write generic functions:

```
1  let rec sum : type a. a -> a repr -> int = fun x -> function
2    | TyInt -> x
3    | r -> sum_ (to_spine x r)
4  and sum_ : type a. a spine -> int = function
5    | Constr _ -> 0
6    | App (k, (x, ty)) -> sum_ k + sum x ty
```

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

```
1  type 'b query = { query : 'a. 'a -> 'a repr -> 'b }
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Generic functions : Scrap your boilerplate (Reloaded)

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Queries are naturally equipped with the following two combinators:

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

```
1  let rec map_query_ : type a b. b query -> a spine -> b list =  
2    fun q -> function  
3      | Constr x -> []  
4      | App (f, (x, ty)) -> map_query_ q f @ [q.query x ty]
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Generic functions : Scrap your boilerplate (Reloaded)

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

```
1  let rec apply_query
2  : type b. (b -> b -> b) -> b query -> b query =
3    fun f q ->
4      { query = fun x r ->
5        reduce f ((q.query x r)
6          :: (map_query (apply_query f q)).query x r)
7      }
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Generic functions : Scrap your boilerplate (Reloaded)

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

We eventually end up with a very succinct definition for the generic sum function:

```
1  let int_query : type a. a -> a repr -> int =  
2    fun x -> function TyInt -> x | r -> 0  
3  
4  let sum : int query =  
5    apply_query ( + ) { query = int_query }
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

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 encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax
Operational semantics
Type System
Metatheory
Implementation of a type checker
for MLG1

Applications

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

GADTs encodings

Bibliography

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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Singleton types

```
1  type zero
2  type 'a succ
3  type 'a nat =
4      | Z : zero nat
5      | S : 'a nat -> ('a 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 `'a succ nat` for some `'a` 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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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:

```
1  type ('a, 'b) add
```

Second, write a suitable specification for **append**:

```
1  let rec append
2  : type n m. n list -> m list -> ((n, m) add) vector =
3  function Nil -> fun v -> v
4  | Cons (x, xs) -> fun v -> Cons (x, append xs v)
```

Is it well-typed?

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

```
1 let rec append
2   : type n m. n list -> m list -> ((n, m) add) vector =
3   function
4   | Nil ->
5     (* n = zero *)
6     fun v -> v
7   | Cons (x, xs) ->
8     (*  $\exists \gamma. n = \gamma \text{ succ}$  *)
9     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 β -equivalence, which forces every type-level computation to always terminate.

Or ...

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

We can reformulate a bit our type definitions:

```
1 type ('a, 'b, 'c) add =  
2 | AddZ : forall 'a.  
3   add (zero, 'a, 'a)  
4 | AddS : forall 'a 'b.  
5   add ('a, 'b, 'c) -> add (succ 'a, 'b, succ 'c)
```

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

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Values of GADTs as proof terms

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

The append function can be updated accordingly:

```
1 let rec append : type n m o.  
2 (n, m, o) add -> ('a, n) list -> ('a, m) list -> ('a, o) list =  
3 function  
4 | AddZ ->  
5   (* n = zero, m = o *)  
6   (function Nil -> fun v -> v)  
7 | AddS p ->  
8   (*  $\exists k.l.n = k \text{ succ}, m = l \text{ succ} *$  *)  
9   (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.

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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:

```
1  type ('a, 'b) eq =  
2    | Refl : ('a, 'a) eq
```

Theorem ([8])

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

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 encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax
Operational semantics
Type System
Metatheory
Implementation of a type checker
for MLG1

Applications

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

GADTs encodings

Bibliography

A representation for type-class dictionaries

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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** [pottier06].

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Type-class dictionary typed by a GADT

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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:

```
1 class Eq (X) { equal : X -> X -> bool }
2 inst Eq (Int) { equal = ... }
3 inst Eq (X) => Eq (list X) { equal = ... }
```

would lead to the following GADT declaration:

```
1 type 'a eq =
2 | EqInt : int eq
3 | EqList : 'a eq -> 'a list eq
```

Where should the code for the class methods be put?

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Interpretation of instance evidence

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

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.

```
1  let rec eq_int x y = (x = y)
2
3  and eq_list dict l1 l2 =
4    match l1, l2 with
5    | [], [] -> true
6    | x :: xs, y :: ys -> equal dict x y
7    | _ -> false
8
9  and equal : eq 'a -> 'a -> 'a -> bool = function
10 | EqInt -> eq_int
11 | EqList dict -> eq_list dict
```

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Interpretation of instance evidence

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

```
1 class Eq (X) => Ord (X) { lt : X -> X -> Int }
```

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

```
1 let rec getEqFromOrd : type a. a ord -> a eq = function  
2 | OrdInt -> EqInt  
3 | OrdList d -> EqList (getEqFromOrd d)
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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 the C type parameter with the same type.

The following pattern matching is exhaustive for this reason:

```
1  let getEqFromEqList : type a. a list eq -> a eq = function
2  | EqList dict -> dict
```

This function realizes the rule « $\forall \alpha. \text{Eq} (\text{List } \alpha) \rightarrow \text{Eq } \alpha$ », 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 « 'a eq » is hidden inside the environment of the closure that built the dictionary « 'a list eq ».

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Plan

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

GADTs encodings

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Let's look for an encoding of GADTs

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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 [9].

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

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.)

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Church-style encoding of natural numbers

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

$$succ \equiv \lambda x. \lambda s. \lambda z. s (x s z)$$

$$zero \equiv \lambda s. \lambda z. z$$

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Church-style encoding of natural numbers

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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 :

$$succ \equiv \lambda(x : \mathbb{N}). \Lambda X. \lambda(s : X \rightarrow X) (z : X). s (x X s z)$$

$$zero \equiv \Lambda X. \lambda(s : X \rightarrow X) (z : X). z$$

$$\mathbb{N} \equiv \forall X. (X \rightarrow X) \rightarrow X \rightarrow X$$

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Type definition for the GADT type constructor “term”

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

term $\alpha \equiv \forall \Theta : \star \Rightarrow \star.$

(for case Lit) $(\text{int} \rightarrow \Theta \text{ int}) \rightarrow$

(for case Pair) $(\forall \beta \gamma. \Theta \beta \rightarrow \Theta \gamma \rightarrow \Theta (\beta \times \gamma)) \rightarrow$

(for case Fst) $(\forall \beta \gamma. \Theta (\beta \times \gamma) \rightarrow \Theta \beta) \rightarrow$

(for case Snd) $(\forall \beta \gamma. \Theta (\beta \times \gamma) \rightarrow \Theta \gamma) \rightarrow$

$\Theta \alpha$

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

Type definition for the GADT type constructor “term”

term $\alpha \equiv \forall \Theta : \star \Rightarrow \star.$

(for case Lit) $(\text{int} \rightarrow \Theta \text{ int}) \rightarrow$

(for case Pair) $(\forall \beta \gamma. \Theta \beta \rightarrow \Theta \gamma \rightarrow \Theta (\beta \times \gamma)) \rightarrow$

(for case Fst) $(\forall \beta \gamma. \Theta (\beta \times \gamma) \rightarrow \Theta \beta) \rightarrow$

(for case Snd) $(\forall \beta \gamma. \Theta (\beta \times \gamma) \rightarrow \Theta \gamma) \rightarrow$

$\Theta \alpha$

What is Θ exactly?

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Type definition for the GADT type constructor “term”

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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

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”

term $\alpha \equiv \forall \Theta : \star \Rightarrow \star.$

(for case Lit) $(\text{int} \rightarrow \Theta \text{ int}) \rightarrow$

(for case Pair) $(\forall \beta \gamma. \Theta \beta \rightarrow \Theta \gamma \rightarrow \Theta (\beta \times \gamma)) \rightarrow$

(for case Fst) $(\forall \beta \gamma. \Theta (\beta \times \gamma) \rightarrow \Theta \beta) \rightarrow$

(for case Snd) $(\forall \beta \gamma. \Theta (\beta \times \gamma) \rightarrow \Theta \gamma) \rightarrow$

$\Theta \alpha$

How to read this type definition?

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Type definition for the GADT type constructor “term”

$\text{term } \alpha \equiv \forall \Theta : \star \Rightarrow \star.$

(for case Lit) $(\text{int} \rightarrow \Theta \text{ int}) \rightarrow$

(for case Pair) $(\forall \beta \gamma. \Theta \beta \rightarrow \Theta \gamma \rightarrow \Theta (\beta \times \gamma)) \rightarrow$

(for case Fst) $(\forall \beta \gamma. \Theta (\beta \times \gamma) \rightarrow \Theta \beta) \rightarrow$

(for case Snd) $(\forall \beta \gamma. \Theta (\beta \times \gamma) \rightarrow \Theta \gamma) \rightarrow$

$\Theta \alpha$

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 $\text{term } \alpha$ 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 $\alpha = \gamma$;

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Type definition for the GADT type constructor “term”

term $\alpha \equiv \forall \Theta : \star \Rightarrow \star.$

(for case Lit) $(\text{int} \rightarrow \Theta \text{ int}) \rightarrow$

(for case Pair) $(\forall \beta \gamma. \Theta \beta \rightarrow \Theta \gamma \rightarrow \Theta (\beta \times \gamma)) \rightarrow$

(for case Fst) $(\forall \beta \gamma. \Theta (\beta \times \gamma) \rightarrow \Theta \beta) \rightarrow$

(for case Snd) $(\forall \beta \gamma. \Theta (\beta \times \gamma) \rightarrow \Theta \gamma) \rightarrow$

$\Theta \alpha$

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Term definition for term data constructor

$\text{lit} : \text{int} \rightarrow \text{term int}$

$\text{lit} = \lambda x : \text{int}. \Lambda \Theta. \lambda \text{lit pair fst snd}. \text{lit } x$

$\text{pair} : \forall \alpha \beta. \text{term } \alpha \rightarrow \text{term } \beta \rightarrow \text{term } (\alpha \times \beta)$

$\text{pair} = \Lambda \alpha. \Lambda \beta. \lambda t : \text{term } \alpha. \lambda u : \text{term } \beta.$

$\Lambda \Theta. \lambda \text{lit pair fst snd}.$

$\text{pair } \alpha \beta$

$(t \ \Theta \ \text{lit pair fst snd})$

$(u \ \Theta \ \text{lit pair fst snd})$

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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Tagless interpreter in F_ω

$\text{eval} : \forall \alpha. \text{term } \alpha \rightarrow \alpha$

$\text{eval} = \Lambda \alpha. \lambda p : \text{term } \alpha.$

$p \ \lambda \alpha. \alpha$

$(\Lambda \alpha. \lambda x. x)$

$(\Lambda \alpha \beta. \lambda x \ y. (x, y))$

$(\Lambda \alpha \beta. \lambda (x, y). x)$

$(\Lambda \alpha \beta. \lambda (x, y). y)$

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.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas

yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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 [10].

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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 :

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Thus, is a primitive notion of GADT really relevant?

`print` : $\forall \alpha. \alpha \rightarrow \text{repr } \alpha \rightarrow \text{string}$

`print` = $\Lambda \alpha. \lambda(x : \alpha). \lambda(r : \text{repr } \alpha).$
 $r ?$

$(\text{string_of_int } ?)$

$(\Lambda \beta. \Lambda \gamma. \lambda(c_1 : \Theta \beta). \lambda(c_2 : \Theta \gamma). ?)$

$(\Lambda \beta. \Lambda \gamma. \lambda(c_1 : \Theta \beta). \lambda(c_2 : \Theta \gamma). ?)$

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas

yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Thus, is a primitive notion of GADT really relevant?

$\text{print} : \forall \alpha. \alpha \rightarrow \text{repr } \alpha \rightarrow \text{string}$
 $\text{print} = \Lambda \alpha. \lambda (x : \alpha). \lambda (r : \text{repr } \alpha). \\ r \ \lambda \alpha. \text{string} \\ (\text{string_of_int } ?) \\ (\Lambda \beta. \Lambda \gamma. \lambda (c_1 : \text{string}. \lambda (c_2 : \Theta \ \gamma). ?) \\ (\Lambda \beta. \Lambda \gamma. \lambda (c_1 : \text{string}. \lambda (c_2 : \Theta \ \gamma). ?))$

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Thus, is a primitive notion of GADT really relevant?

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

```
print  :   $\forall \alpha. \alpha \rightarrow \text{repr } \alpha \rightarrow \text{string}$ 
print  =   $\Lambda \alpha. \lambda (x : \alpha). \lambda (r : \text{repr } \alpha).$ 
            $(r \ \lambda \alpha. \alpha \rightarrow \text{string}$ 
              $(\text{string\_of\_int } )$ 
              $(\Lambda \beta. \Lambda \gamma. \lambda (c_1 : \beta \rightarrow \text{string}). \lambda (c_2 : \gamma \rightarrow \text{string}).$ 
                $\lambda (x' : \beta + \gamma). \dots)$ 
              $(\Lambda \beta. \Lambda \gamma. \lambda (c_1 : \beta \rightarrow \text{string}). \lambda (c_2 : \gamma \rightarrow \text{string}).$ 
                $\lambda (x' : \beta \times \gamma). \dots)) \ x$ 
```

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Thus, is a primitive notion of GADT really relevant?

$$\begin{aligned}\text{print} & : \quad \forall \alpha. \alpha \rightarrow \text{repr } \alpha \rightarrow \text{string} \\ \text{print} & = \quad \Lambda \alpha. \lambda(x : \alpha). \lambda(r : \text{repr } \alpha). \\ & \quad (r \ \lambda \alpha. \alpha \rightarrow \text{string} \dots) \ x\end{aligned}$$

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

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

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 :

```
1  Fixpoint print (A : Type) (x : A) (r : repr A) : string :=
2    (match r in repr T return (T -> string) with
3      | TyNat => fun (x : nat) => string_of_nat x
4      | TySum B C ty1 ty2 => fun (x : B + C) =>
5        match x with
6          | inl x1 => print B x1 ty1
7          | inr x2 => print C x2 ty2
8        end
9      | TyProd B C ty1 ty2 => fun (x : B * C) =>
10        append (print B (fst x) ty1) (print C (snd x) ty2)
11    end) x.
```

Rich types,
Tractable typing
- Dependently-typed
programming
languages -

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming


Dependently-typed programming

Typed intermediate languages


GADTs encodings

Bibliography


Bibliography I

 Leijen, D. & Meijer, E. Domain specific embedded compilers. in Proceedings of the 2nd conference on Domain-specific languages (ACM, Austin, Texas, United States, 1999), 109–122. isbn: 1-58113-255-7.

<http://doi.acm.org/10.1145/331960.331977>
(cit. on p. 7).

 Liskov, B. & Zilles, S. Programming with abstract data types. in Proceedings of the ACM SIGPLAN symposium on Very high level languages (ACM, Santa Monica, California, United States, 1974), 50–59.

<http://doi.acm.org/10.1145/800233.807045>
(cit. on p. 9).

 Xi, H., Chen, C. & Chen, G. Guarded Recursive Datatype Constructors. in Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages (New Orleans, Jan. 2003), 224–235 (cit. on p. 20).

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

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

Embedding domain-specific
languages

Generic programming


Dependently-typed programming


Typed intermediate languages


GADTs encodings


Bibliography

Bibliography II

 Cheney, J. & Hinze, R. First-class phantom types. Tech. rep. (Cornell University, 2003) [cit. on p. 20].

 Läuffer, K. & Odersky, M. An Extension of ML with First-Class Abstract Types. in ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California (June 1992), 78–91 [cit. on p. 38].

 Carette, J., Kiselyov, O. & Shan, C.-c. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**, 509–543. issn: 0956-7968 (2009) [cit. on p. 85].

 Danvy, O. Functional unparsing. J. Funct. Program. **8**, 621–625. issn: 0956-7968. <http://dl.acm.org/citation.cfm?id=969600.969603> (6 Nov. 1998) [cit. on p. 89].

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming





Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Bibliography III

-  Johann, P. & Ghani, N. Foundations for structured programming with GADTs. in POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (ACM, San Francisco, California, USA, 2008), 297–308. isbn: 978-1-59593-689-9 (cit. on p. 104).
-  Pfenning, F. & Lee, P. LEAP: A Language with Eval And Polymorphism. (Cit. on p. 114).
-  Mandelbaum, Y. & Stump, A. GADTs for the OCaml Masses. Tech. rep. () (cit. on p. 125).
-  Schrijvers, T., Peyton Jones, S., Sulzmann, M. & Vytiniotis, D. Complete and decidable type inference for GADTs. in Proceedings of the 14th ACM SIGPLAN international conference on Functional programming (ACM, Edinburgh, Scotland, 2009), 341–352. isbn: 978-1-60558-332-7.
<http://doi.acm.org/10.1145/1596550.1596599>.

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation
of Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming




Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography

Bibliography IV

-  Pottier, F. & Régis-Gianas, Y. Stratified type inference for generalized algebraic data types. in POPL (eds Morrisett, J. G. & Jones, S. L. P.) (ACM, 2006), 232–244. isbn: 1-59593-027-2.
-  Vytiniotis, D., Jones, S. L. P., Schrijvers, T. & Sulzmann, M. OutsideIn(X) Modular type inference with local assumptions. J. Funct. Program. **21**, 333–412 (2011).
-  Simonet, V. & Pottier, F. A Constraint-Based Approach to Guarded Algebraic Data Types. ACM Transactions on Programming Languages and Systems. To appear (Dec. 2005).

Rich types,
Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Informal presentation of
Generalized
Algebraic Data types

Formal presentation of
GADTs

Syntax

Operational semantics

Type System

Metatheory

Implementation of a type checker
for MLG1

Applications

Embedding domain-specific
languages

Generic programming

Dependently-typed programming

Typed intermediate languages

GADTs encodings

Bibliography