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

Yann Régis-Gianas
yrg@irif.fr

2017-12-22

Plan

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

GADTs encodings

Type inference in ML extended with GADTs

Yann Régis-Gianas
yrg@irif.fr

GADTs through a motivating example

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

Yann Régis-Gianas
yrg@irif.fr

Algebraic data types : data constructors

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.

Algebraic data types : pattern matching

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

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

```
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
```


Yann Régis-Gianas
yrg@irif.fr

A safer head extraction?

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

Yann Régis-Gianas
yrg@irif.fr

A trick in vanilla ML : Phantom types [LM99]

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 `'a t` will not represent the type of a component of `v` but a static property attached to `v`.

Yann Régis-Gianas
yrg@irif.fr

A phantom type for emptiness

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
```


Yann Régis-Gianas
yrg@irif.fr

Smart constructors to define a phantom type

By turning this type into an abstract data type [LZ74], 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
```


Yann Régis-Gianas
yrg@irif.fr

A safer (?) head extraction

```
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.

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.

Yann Régis-Gianas
yrg@irif.fr

Generalized Algebraic Data Types

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

Yann Régis-Gianas
yrg@irif.fr

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**.

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`.

Automatic type refinement in action

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

```
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 `l` has both the type `a glist` and the type `nonempty glist`** because `l` has been built by application of the data constructor **`Cons`**. The second type is enough to accept the application of `hd` on `l`.

Yann Régis-Gianas
yrg@irif.fr

Where does the magic lie?

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

Yann Régis-Gianas
yrg@irif.fr

Type equalities

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).)

Yann Régis-Gianas
yrg@irif.fr

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.

Yann Régis-Gianas
yrg@irif.fr

Type-checking `totalHd`

```
1  let totalHd : type a. a glist -> int = function
2    | Nil -> 42
3    | (Cons _) as l -> hd l
4      (* Here, a = nonempty. *)
5      (* As Γ ⊢ l : a glist *)
6      (* and a = nonempty ⊨ a glist = nonempty glist, *)
7      (* by conversion, Γ ⊢ l : nonempty glist *)
```


Yann Régis-Gianas
yrg@irif.fr

Origins of GADTs

Generalized Algebraic Data types have been first introduced independently by Hongwei Xi, Chiyan Chen and Gang Chen [XCC03] and by James Cheney and Ralf Hinze [CH03].

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.

Yann Régis-Gianas
yrg@irif.fr

Exercise

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.

Yann Régis-Gianas
yrg@irif.fr

Exercises

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?

Yann Régis-Gianas
yrg@irif.fr

Exercises

③ Examine the following function:

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

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

Yann Régis-Gianas
yrg@irif.fr

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

Yann Régis-Gianas
yrg@irif.fr

MLGI

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

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

Yann Régis-Gianas
yrg@irif.fr

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

Syntax of MLGI

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

Syntax of MLGI (continued)

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

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

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

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

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

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

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

Yann Régis-Gianas
yrg@irif.fr

Values

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

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

Evaluation context

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

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

Reduction rules

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

$$\begin{array}{rl}
\texttt{(E-Match)} & \mathsf{match}\, K\,(v_1, \ldots, v_n)\, \mathsf{with}\, K\ (x_1, \ldots, x_n) \Rightarrow t \mid \overline{c} \\
\to & [x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]t
\end{array}$$

$$\begin{array}{rl}
\texttt{(E-NoMatch)} & \mathsf{match}\, K'\,(v_1, \ldots, v_m)\, \mathsf{with}\, K\ (x_1, \ldots, x_n) \Rightarrow t \mid \overline{c} \\
\to & \mathsf{match}\, K'\,(v_1, \ldots, v_m)\, \mathsf{with}\, \overline{c} \\
 & \text{if } K \neq K'
\end{array}$$

Stuck terms

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

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

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

Yann Régis-Gianas
yrg@irif.fr

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

Yann Régis-Gianas
yrg@irif.fr

Syntax for types

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

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

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

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

Specification of data constructors

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

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

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

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

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

Yann Régis-Gianas
yrg@irif.fr

GADTs includes ADTs

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

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

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

Example

Here is two data constructor declarations for polymorphic lists:

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

GADTs includes iso-existential data types

The type variables $\overline{\beta}$ are said to be local because when the set of generalized type parameters is empty, the $\overline{\beta}$ do not escape through the output type. As a consequence, they can be used to implement iso-existential types [LO92].

Example

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

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

Yann Régis-Gianas
yrg@irif.fr

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:

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

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

Yann Régis-Gianas
yrg@irif.fr

Syntax for type equalities

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

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

Judgments

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

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

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

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

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

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

Well-typed terms

Definition (Well-typed terms)

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

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

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

Yann Régis-Gianas
yrg@irif.fr

Rule for data constructor applications

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

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.

Yann Régis-Gianas
yrg@irif.fr

Conversion rule

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

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

$$\frac{}{E_1 \wedge \tau_1 = \tau_2 \wedge E_2 \models \tau_1 = \tau_2} \qquad \frac{E \models \tau_1 = \tau_1' \qquad E \models \tau_2 = \tau_2'}{E \models \tau_1 \to \tau_2 = \tau_1' \to \tau_2'}$$

$$\frac{E \models \overline{\tau}_1 = \overline{\tau}_1' \qquad E \models \overline{\tau}_2 = \overline{\tau}_2'}{E \models \varepsilon \overline{\tau}_1 \overline{\tau}_2 = \varepsilon \overline{\tau}_1' \overline{\tau}_2'} \qquad \frac{}{E \models \tau = \tau} \qquad \frac{E \models \tau_2 = \tau_1}{E \models \tau_1 = \tau_2}$$

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

Rule for pattern matchings

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

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

Rule for clauses

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

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

Extraction of type equalities

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

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

Specialization of `Pat` to regular ADT

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

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

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

Specialization of `Pat` to iso-existential ADT

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

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

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

Back to the general version

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

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

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

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

Yann Régis-Gianas
yrg@irif.fr

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

Yann Régis-Gianas
yrg@irif.fr

Metatheory

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

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

Yann Régis-Gianas
yrg@irif.fr

Key lemma

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

Lemma

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

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

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

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

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

Yann Régis-Gianas
yrg@irif.fr

Subject reduction

Theorem (Subject reduction)

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

Proof.

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

Yann Régis-Gianas
yrg@irif.fr

Progress

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

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

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

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

Theorem (Progress)

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

Improving progress thanks to dead code elimination

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

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

Yann Régis-Gianas
yrg@irif.fr

Improving progress thanks to dead code elimination

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**.

Improving progress thanks to dead code elimination

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

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

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.

Yann Régis-Gianas
yrg@irif.fr

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

Yann Régis-Gianas
yrg@irif.fr

Type-checking algorithm

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

Fortunately, the problem of **checking that a term 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.

Yann Régis-Gianas
yrg@irif.fr

An algorithm based on congruence closure

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

1. At every pattern matching branches, compute type equalities E by confronting the type of the scrutinee with the type of the pattern. Two sub-cases:
   - If E is consistent, then convey the local type equalities E along the type-checking of the case branch.
   - If E is inconsistent, then the case branch is dead code. [2]
2. At every point where two types must be equal, decide this type equality modulo E using congruence closure [**Nelson80**].

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

An algorithm based on eager application of MGU

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

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

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

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

Yann Régis-Gianas
yrg@irif.fr

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

Yann Régis-Gianas
yrg@irif.fr

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.

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

Yann Régis-Gianas
yrg@irif.fr

Example : A language of expressions

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

```
1  (* t ::= 0, 1, ... | π1 t | π2 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
```


Yann Régis-Gianas
yrg@irif.fr

A running example

An interpreter of this language is a straightforward inductive function:

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

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

Yann Régis-Gianas
yrg@irif.fr

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  (* ∀eT, Pre:" ⊢ e : T'' ⇒ Post:" ⊢ eval e : T'' *)
4  let rec eval : term -> value = function
5    | Lit x -> VInt x
6      (* By inversion of Pre, we have ⊢ Lit x : int *)
7      (* Thus, T = int and, we indeed have, ⊢ VInt x : int *)
8    | Pair (t1, t2) ->  VPair (eval t1, eval t2)
9      (* By inversion of Pre, there exist β1, β2 such that *)
10     (* ⊢ Pair (t1, t2) : β1 × β2, ⊢ t1 : β1, and ⊢ t2 : β2. *)
11     (* Thus, ⊢ eval t1 : β1, and ⊢ eval t2 : β2. *)
12     (* Eventually, we obtain ⊢ (eval t1, eval t2) : β1 × β2. *)
13   | Fst t -> ...
14   | Snd t -> ...
```

Example : A commented interpreter for expressions

```
1  (* [eval e] interprets a well-typed expression of type T *)
2  (* into a value of type T, or more formally: *)
3  (* ∀eT, Pre:" ⊢ e : T'' ⇒ Post:" ⊢ 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, ∃β,⊢ t : T × β *)
11     (* Then, ⊢ eval t : T × β *)
12     (* And by the classification lemma, eval t = (v1, v2) *)
13     (* with ⊢ v1 : T, which is what we expected. *)
14   | Snd t -> ...
```


Yann Régis-Gianas
yrg@irif.fr

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, eval t = (v1, v2) *)
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**.

How to encode a predicate using a type?

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

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

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

Yann Régis-Gianas
yrg@irif.fr

Encoding a predicate using a type

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

Yann Régis-Gianas
yrg@irif.fr

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  (* τ ::= int | τ × τ    *)
2  type int_type
3  type ('a, 'b) pair_type
```

A language of **well-typed** expressions

```
1  (* t ::= 0, 1, ... | π1 t | π2 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`**.

Yann Régis-Gianas
yrg@irif.fr

An interpreter for well-typed expressions

```
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      (* ∃ b c. a = (b, c) pair_ty, t1 : b term, t2 : c term *)
7      VPair (eval t1, eval t2)
8    | Fst t ->
9      (* ∃ b. t : ((a, b) pair_ty) term. *)
10     (match eval t with VPair (v1, _) -> v1)
11   | Snd t ->
12     (* ∃ b. t : ((b, a) pair_ty) term. *)
13     (match eval t with VPair (_, v2) -> v2)
```


Yann Régis-Gianas
yrg@irif.fr

Type equivalence modulo type equalities

Following the usual ML typing rules, we know that

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

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).

Yann Régis-Gianas
yrg@irif.fr

An interpreter for well-typed expressions

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


Yann Régis-Gianas
yrg@irif.fr

Tagless interpreter

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

Thus, we can safely remove it by defining "`type '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**.

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

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

```
1  (* t ::= 0, 1, ... | π1 t | π2 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
```

Example : A tagless interpreter for well-typed expressions

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


Yann Régis-Gianas
yrg@irif.fr

Exercise

Exercise 1:

Write a tagless interpreter for simply typed lambda calculus.

Yann Régis-Gianas
yrg@irif.fr

λ-term with de Bruijn indices

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

```
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
```

A tagless interpreter for simply typed λ-calculus

```
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)
```


Yann Régis-Gianas
yrg@irif.fr

Digression

Can a tagless interpreter be implemented differently?

Yann Régis-Gianas
yrg@irif.fr

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 [CKS09].

Yann Régis-Gianas
yrg@irif.fr

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

Yann Régis-Gianas
yrg@irif.fr

Dynamic representation of types

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
```

Dynamic representation of types

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        (* ``α = int'', so ``x : int''. *)
6        string_of_int x
7      | TySum (ty1, ty2) ->
8        (* ``∃βγ. α = β + γ'', so ``x : β + γ''*)
9        (match x with
10         | Left x1 -> "L" ^ print x1 ty1
11         | Right x2 -> "R" ^ print x2 ty2)
12     | TyProd (ty1, ty2) ->
13       (* ``∃βγ. α = β * γ'', so ``x : β * γ''*)
14       "(" ^ print (fst x) ty1 ^ ", "^ print (snd x) ty2 ^")"
```


Yann Régis-Gianas
yrg@irif.fr

Printf

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 [Dan98].

Yann Régis-Gianas
yrg@irif.fr

Generic functions : Scrap your boilerplate (Reloaded)

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

```
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
```

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
```

Generic functions : Scrap your boilerplate (Reloaded)

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 }
```


Yann Régis-Gianas
yrg@irif.fr

Generic functions : Scrap your boilerplate (Reloaded)

Queries are naturally equipped with the following two combinators:

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

```
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]
```


Yann Régis-Gianas
yrg@irif.fr

Generic functions : Scrap your boilerplate (Reloaded)

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

```
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      }
```


Yann Régis-Gianas
yrg@irif.fr

Generic functions : Scrap your boilerplate (Reloaded)

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 }
```


Yann Régis-Gianas
yrg@irif.fr

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

Yann Régis-Gianas
yrg@irif.fr

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.

Yann Régis-Gianas
yrg@irif.fr

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.

Yann Régis-Gianas
yrg@irif.fr

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?

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

```
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    (* ∃γ.n = γ succ *)
9    fun v -> Cons (x, append xs v)
```

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

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

Yann Régis-Gianas
yrg@irif.fr

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 ...

Yann Régis-Gianas
yrg@irif.fr

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

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.

Yann Régis-Gianas
yrg@irif.fr

Values of GADTs as proof terms

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    (* ∃kl.n = k succ, m = l 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.

Yann Régis-Gianas
yrg@irif.fr

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 ([JG08])

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

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

Yann Régis-Gianas
yrg@irif.fr

A representation for type-class dictionaries

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

Dictionaries are usually represented as records. Another representation is based on a GADT. This GADT-based transformation is called **concretization** [**pottier06**].

Yann Régis-Gianas
yrg@irif.fr

Type-class dictionary typed by a GADT

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

Yann Régis-Gianas
yrg@irif.fr

Type-class dictionary typed by a GADT

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

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

would lead to the following GADT declaration:

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

Where should the code for the class methods be put?

Yann Régis-Gianas
yrg@irif.fr

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
```


Yann Régis-Gianas
yrg@irif.fr

Interpretation of instance evidence

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

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

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

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

More powerful elaboration

The « non overlapping instance definitions » property can be rephrased in the translated GADT declarations: for each GADT C that reflects a class C, there cannot be two data constructors which instantiate 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 « $\forall\alpha.\mathrm{Eq}\ (\mathrm{List}\ \alpha) \to \mathrm{Eq}\ \alpha$ », 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` ».

Yann Régis-Gianas
yrg@irif.fr

Plan

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

GADTs encodings

Type inference in ML extended with GADTs

Yann Régis-Gianas
yrg@irif.fr

Let's look for an encoding of GADTs

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

Yann Régis-Gianas
yrg@irif.fr

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[PL].

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

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

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

Church-style encoding of natural numbers

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

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

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

Yann Régis-Gianas
yrg@irif.fr

Church-style encoding of natural numbers

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

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

$$
\begin{array}{rcl}
succ & \equiv & \lambda(x : \mathbb{N}).\Lambda X.\lambda(s : X \to X)\ (z : X).s\ (x\ X\ s\ z) \\
zero & \equiv & \Lambda X.\lambda(s : X \to X)\ (z : X).z \\
\mathbb{N} & \equiv & \forall X.(X \to X) \to X \to X
\end{array}
$$

Yann Régis-Gianas
yrg@irif.fr

Type definition for the GADT type constructor "term"

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

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

Yann Régis-Gianas
yrg@irif.fr

Type definition for the GADT type constructor "term"

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

What is Θ exactly?

Yann Régis-Gianas
yrg@irif.fr

Type definition for the GADT type constructor "term"

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

What is Θ exactly?

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

Yann Régis-Gianas
yrg@irif.fr

Type definition for the GADT type constructor "term"

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

How to read this type definition?

Yann Régis-Gianas
yrg@irif.fr

Type definition for the GADT type constructor "term"

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

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

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

(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 $\alpha = \gamma$;

Type definition for the GADT type constructor "term"

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

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

Yann Régis-Gianas
yrg@irif.fr

Term definition for term data constructor

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

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

Exercise: Write the F_3 term for fst.

Tagless interpreter in F_ω

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

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

Yann Régis-Gianas
yrg@irif.fr

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 [MS].

Yann Régis-Gianas
yrg@irif.fr

Thus, is a primitive notion of GADT really relevant?

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

For instance, our second example turns into :

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

Yann Régis-Gianas
yrg@irif.fr

Thus, is a primitive notion of GADT really relevant?

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

Yann Régis-Gianas
yrg@irif.fr

Thus, is a primitive notion of GADT really relevant?

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

Yann Régis-Gianas
yrg@irif.fr

Thus, is a primitive notion of GADT really relevant?

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

Yann Régis-Gianas
yrg@irif.fr

Thus, is a primitive notion of GADT really relevant?

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

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

Yann Régis-Gianas
yrg@irif.fr

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.
```


Yann Régis-Gianas
yrg@irif.fr

Plan

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

GADTs encodings

Type inference in ML extended with GADTs

- Why is type inference difficult?
- Stratified type inference
- `OutSideIn(X)`

Yann Régis-Gianas
yrg@irif.fr

Plan

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

GADTs encodings

Type inference in ML extended with GADTs

Why is type inference difficult?

Stratified type inference

`OutSideIn(X)`

Yann Régis-Gianas
yrg@irif.fr

Dealing with a difficult type inference problem

- Type inference with GADTs is difficult and is still an open problem.
- The main reason for that difficulty is **the lack of ML principal types**.
- The purpose of this lecture is to explain the practical **trade-offs** that have been made to smoothly integrate GADTs in ML type inference engines:
  1. The OutsideIn approach [Sch+09].
  2. The Stratified Type Inference approach [PR06].
  3. The modular OutsideIn(X) without let-generalization approach [Vyt+11]

The loss of principality

Let us consider the following function that implements a safe type cast from the type α to the type β, if a proof of $\alpha = \beta$ is provided :

```
1  let cast : ? =
2    fun p x ->
3      match p with
4      | Refl -> x
```


Yann Régis-Gianas
yrg@irif.fr

The loss of principality

Let us consider the following function that implements a safe type cast from the type α to the type β, if a proof of $\alpha = \beta$ is provided :

```
1  let cast : ? =
2    fun p x ->
3      match p with
4      | Refl -> x
```

This program admits at least three **incomparable** type schemes:

- $\forall \alpha \beta.\, \mathrm{eq}\ \alpha\ \beta \to \alpha \to \beta$
- $\forall \alpha \beta.\, \mathrm{eq}\ \alpha\ \beta \to \beta \to \alpha$
- $\forall \alpha \beta \gamma.\, \mathrm{eq}\ \alpha\ \beta \to \gamma \to \gamma$

Yann Régis-Gianas
yrg@irif.fr

Problem 1: What are the available type equalities?

```
1  let cast : ? =
2    fun p x ->
3      match p with
4      | Refl -> x
```

During the type inference process, the types of the sub-expressions are (at least partially) unknown.

In particular, given that we do not know the type of the scrutinee `p`, it is not possible to extract the type refinement of each branch.

Problem 2: How are these type equalities used?

```
1  let cast : ? =
2    fun p x ->
3      match p with
4      | Refl -> x
```

Assume that, at some point, the type `eq 'a 'b` of `p` is known to the type inference engine. This implies that the type equality `'a = 'b` is available in the typing derivation of the expression « `x` ». The type inference engine must consider two different scenarios depending on the use that is made of this equality.

Problem 2: How are these type equalities used?

```
1  let cast : ? =
2    fun p x ->
3      match p with
4      | Refl -> x
```

On the one hand, if we assume that this type equality is used, it is equivalent, **locally**, to assign the type `'a` or the type `'b` to the expression `x`. Yet, outside the pattern matching, the type equality is not available anymore. Thus, we have to be aware that this local choice, if uncontrolled, may have some global effects on the final inferred type for `cast`. Indeed, the type of `x` is the input type of `cast`. Moreover, the output type of `cast` is also contrived by the type of `x`.

Problem 2: How are these type equalities used?

```
1  let cast : ? =
2    fun p x ->
3      match p with
4      | Refl -> x
```

On the other hand, if we assume that this type equality is not used, the type that is ascribed to `x` locally is the same outside the pattern matching and is not related the α and β.

Yann Régis-Gianas
yrg@irif.fr

Problem 3: Is that dead code or not?

```
1  let succ : ? =
2    fun r x ->
3      1 + match r with
4      | TyInt -> x
5      | TyString -> x
```

If we assume that the branch for **`TyString`** is dead then the type **`repr int -> int`** is correct. The second branch cannot be alive because this program would be ill-typed otherwise.

Yann Régis-Gianas
yrg@irif.fr

Four inter-connected problems

A type inference engine for ML with GADTs must deal simultaneously with the following interconnected 4 questions:

- What are the types of the sub-expressions?
- What are the available type equalities?
- How are the type equalities used?
- Is a branch dead?

We need a declarative way of capturing the interrelated typing constraints.

Implication typing constraints

As usual, we will reduce the problem of type inference to the satisfiability of typing constraints. Standard conjunctions of equalities under a mixed prefix, that are sufficient for ML type inference, are not expressive enough to denote what is happening with local type equalities. We need an additional construction to capture the process of adding some type equalities as local assumptions of some specific typing constraints. That is exactly the purpose of the **implication** connective.

We extend the syntax of typing constraints (for type inference):

$$C ::= \ldots \mid C \Rightarrow C$$

Yann Régis-Gianas
yrg@irif.fr

New constraint generation rules

Recall the rules for ML:

$$
\begin{array}{rcl}
[\![x : \tau]\!] & = & x \preceq \tau \\
[\![\lambda x.T : \tau]\!] & = & \exists \gamma_1 \gamma_2 . \tau = \gamma_1 \to \gamma_2 \wedge \text{def } x : \gamma_1 \text{ in } [\![T : \gamma_2]\!] \\
[\![T_1\ T_2 : \tau]\!] & = & \exists \gamma . [\![T_1 : \gamma \to \tau]\!] \wedge [\![T_2 : \gamma]\!] \\
[\![\text{let } x = T_1 \text{ in } T_2 : \tau]\!] & = & \text{def } x : \forall \gamma [[\![T_1 : \gamma]\!] . \gamma \text{ in } [\![T_2 : \tau]\!]
\end{array}
$$

What would be the constraint generation rule for pattern matching?

Yann Régis-Gianas
yrg@irif.fr

New constraint generation rules

$$[\![\text{match } T \text{ with } p_1 \Rightarrow T_1 \mid \ldots \mid p_n \Rightarrow T_n : \tau]\!] =$$

Yann Régis-Gianas
yrg@irif.fr

New constraint generation rules

$$[\![\text{match } T \text{ with } p_1 \Rightarrow T_1 \mid \ldots \mid p_n \Rightarrow T_n : \tau]\!] =$$
$$\exists\gamma.\ [\![T : \gamma]\!] \qquad \text{The scrutinee has type } \gamma.$$

New constraint generation rules

$$[\![\text{match } T \text{ with } p_1 \Rightarrow T_1 \mid \ldots \mid p_n \Rightarrow T_n : \tau]\!] =$$

$$\exists\gamma.\ [\![T : \gamma]\!] \qquad \text{The scrutinee has type } \gamma.$$

$$\wedge\ ([\![p_1 : \gamma]\!] \qquad \text{The pattern has type } \gamma.$$

Yann Régis-Gianas
yrg@irif.fr

New constraint generation rules

$$
\begin{array}{lr}
[\![\mathsf{match}\ T\ \mathsf{with}\ p_1 \Rightarrow T_1 \mid \ldots \mid p_n \Rightarrow T_n : \tau]\!] & = \\
\quad \exists \gamma.\ [\![T : \gamma]\!] & \text{The scrutinee has type } \gamma. \\
\quad\quad \wedge\ ([\![p_1 : \gamma]\!] & \text{The pattern has type } \gamma. \\
\quad\quad\quad \wedge\ \forall \overline{\beta}_1.\ \mathsf{def}\ \Gamma_1\ \mathsf{in}\ C_1 \Rightarrow & \text{The local assumptions...}
\end{array}
$$

A call to an auxiliary function $[\![p_i \uparrow \gamma]\!]$ defines what the $\overline{\beta}_i$, Γ_i and C_i are.

New constraint generation rules

$$
\begin{array}{lr}
[\![\mathsf{match}\ T\ \mathsf{with}\ p_1 \Rightarrow T_1 \mid \ldots \mid p_n \Rightarrow T_n : \tau]\!] & = \\
\quad \exists \gamma.\ [\![T : \gamma]\!] & \text{The scrutinee has type } \gamma. \\
\quad\quad \wedge\ ([\![p_1 : \gamma]\!] & \text{The pattern has type } \gamma. \\
\quad\quad\quad \wedge\ \forall \overline{\beta}_1.\ \mathsf{def}\ \Gamma_1\ \mathsf{in}\ C_1 \Rightarrow & \text{The local assumptions...} \\
\quad\quad\quad\quad [\![T_1 : \tau]\!]) & \text{...under which the body is typed.}
\end{array}
$$

A call to an auxiliary function $[\![p_i \uparrow \gamma]\!]$ defines what the $\overline{\beta}_i$, Γ_i and C_i are.

Yann Régis-Gianas
yrg@irif.fr

New constraint generation rules

$$
\begin{aligned}
&[\![\text{match } T \text{ with } p_1 \Rightarrow T_1 \mid \ldots \mid p_n \Rightarrow T_n : \tau]\!] \quad = \\
&\quad \exists\gamma.\ [\![T : \gamma]\!] && \text{The scrutinee has type } \gamma. \\
&\qquad \wedge\ ([\![p_1 : \gamma]\!] && \text{The pattern has type } \gamma. \\
&\qquad\quad \wedge\ \forall\overline{\beta}_1.\ \text{def } \Gamma_1 \text{ in } C_1 \Rightarrow && \text{The local assumptions...} \\
&\qquad\qquad [\![T_1 : \tau]\!]) && \text{...under which the body is typed.} \\
&\ldots \\
&\qquad \wedge\ ([\![p_n : \gamma]\!] \wedge\ \forall\overline{\beta}_n.\ \text{def } \Gamma_n \text{ in } C_n \Rightarrow [\![T_n : \tau]\!])
\end{aligned}
$$

A call to an auxiliary function $[\![p_i \uparrow \gamma]\!]$ defines what the $\overline{\beta}_i, \Gamma_i$ and C_i are.

Yann Régis-Gianas
yrg@irif.fr

How to extract the local assumptions?

If data constructors are declared using constrained type schemes, we get:

$$[\![K\ (x_1, \ldots, x_n) \uparrow \tau]\!] = (\overline{\alpha}\overline{\beta}, \varepsilon\ \overline{\alpha} = \tau \wedge C, (x_1 : \tau_1, \ldots, x_n : \tau_n))$$

where $K :: \forall \overline{\alpha}\overline{\beta}[C].\ \tau_1 \times \ldots \tau_n \to \varepsilon\ \overline{\alpha}$

The equality "$\varepsilon\ \overline{\alpha} = \tau$" is used to instantiate the constraint C with respect to the type of the scrutinee. If "$\varepsilon\ \overline{\alpha} = \tau \wedge C \equiv \mathsf{false}$" then we can conclude that this branch is dead.

Yann Régis-Gianas
yrg@irif.fr

How to extract the local assumptions?

In our formulation, data constructors are declared using ML type schemes:

$$[\![K\ (x_1, \ldots, x_n) \uparrow \tau]\!] = (\overline{\alpha}\overline{\beta}, \varepsilon\ \overline{\alpha}\ \overline{\tau'} = \tau, (x_1 : \tau_1, \ldots, x_n : \tau_n))$$

where $K :: \forall \overline{\alpha}\overline{\beta}.\ \tau_1 \times \ldots \tau_n \to \varepsilon\ \overline{\alpha}\ \overline{\tau'}$

Yann Régis-Gianas
yrg@irif.fr

What is the typing constraint of a pattern?

If data constructors are declared using a constrained type scheme, we get:

$$[\![K\ (x_1, \ldots, x_n) : \tau]\!] = \exists \overline{\alpha}.\ (\tau = \varepsilon\ \overline{\alpha})$$

where $K :: \forall \overline{\alpha}\overline{\beta}[C].\ \tau_1 \times \ldots \tau_n \to \varepsilon\ \overline{\alpha}$

This constraint only checks that the pattern is compatible with the type of the scrutinee to ensure that the pattern matching is well-typed.

Yann Régis-Gianas
yrg@irif.fr

What is the typing constraint of a pattern?

In our formulation, we have:

$$[\![K\ (x_1, \ldots, x_n) : \tau]\!] = \exists \overline{\alpha}\, \overline{\alpha'}.\ (\tau = \varepsilon\ \overline{\alpha}\ \overline{\alpha'})$$

where $K :: \forall \overline{\alpha}\overline{\beta}.\ \tau_1 \times \ldots \tau_n \to \varepsilon\ \overline{\alpha}\ \overline{\tau'}$

Yann Régis-Gianas
yrg@irif.fr

Example

```
1  let f =
2    fun x r ->
3      match r with
4      | TyInt -> string_of_int x
5      | TySum (ty1, ty2) ->
6        (match x with Left x1 -> "L" | Right x2 -> "R")
7      | TyProd (ty1, ty2) ->  "P"
```

A correct typing constraint for `f` would look like (after some simplifications):

$$
\begin{aligned}
&\text{let } f : \forall\gamma.[\exists\gamma_x\gamma_r.\\
&\qquad \gamma = \gamma_x \to \text{repr } \gamma_r \to \text{string}\\
&\quad \wedge\ \gamma_r = \text{int} \Rightarrow \gamma_x = \text{int}\\
&\quad \wedge\ \forall\beta\alpha.\gamma_r = \alpha + \beta \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 + \gamma_2\\
&\quad \wedge\ \forall\beta\alpha.\gamma_r = \alpha \times \beta \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 \times \gamma_2\\
&].\gamma\\
&\text{ in true}
\end{aligned}
$$

How to solve such an implication constraint?

Yann Régis-Gianas
yrg@irif.fr

Solving an implication constraint by hand

$$
\begin{array}{l}
\mathsf{let}\ f : \forall\gamma.[\\
\exists\gamma_x\gamma_r. \\
\qquad \gamma = \gamma_x \to \mathsf{repr}\ \gamma_r \to \mathsf{string} \\
\quad \wedge\ \gamma_r = \mathsf{int} \Rightarrow \gamma_x = \mathsf{int} \\
\quad \wedge\ \forall\beta\alpha.\gamma_r = \alpha + \beta \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 + \gamma_2 \\
\quad \wedge\ \forall\beta\alpha.\gamma_r = \alpha \times \beta \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 \times \gamma_2 \\
].\gamma \\
\ \mathsf{in}\ \mathsf{true}
\end{array}
$$

Clearly, we cannot use first order unification alone to find an assignment of γ_x that satisfies the right hand side of each implication.
One must use the local assumptions, that only refer to γ_r.
Thus, a first step is to unify γ_x and γ_r.

Yann Régis-Gianas
yrg@irif.fr

Solving an implication constraint by hand

$$
\begin{aligned}
&\text{let } f : \forall\gamma.[\\
&\exists\gamma_{xr}.\\
&\qquad \gamma = \gamma_{xr} \to \text{repr } \gamma_{xr} \to \text{string}\\
&\quad \wedge\ \gamma_{xr} = \text{int} \Rightarrow \gamma_{xr} = \text{int}\\
&\quad \wedge\ \forall\beta\alpha.\gamma_{xr} = \alpha + \beta \Rightarrow \exists\gamma_1\gamma_2.\gamma_{xr} = \gamma_1 + \gamma_2\\
&\quad \wedge\ \forall\beta\alpha.\gamma_{xr} = \alpha \times \beta \Rightarrow \exists\gamma_1\gamma_2.\gamma_{xr} = \gamma_1 \times \gamma_2\\
&].\gamma\\
&\ \text{in true}
\end{aligned}
$$

Then, it is tempting to apply the following simplification rules on each implication $C_1 \Rightarrow C_2$:

1. Compute ϕ, the most general unifier of C_1.
2. Rewrite $C_1 \Rightarrow C_2$ into $\phi(C_2)$.

Yet, we will see that this simplification rule is not meaning preserving. (It is sound but not complete.)

Yann Régis-Gianas
yrg@irif.fr

Solving an implication constraint by hand

$$
\begin{aligned}
&\mathsf{let}\ f : \forall\gamma.[\\
&\exists\gamma_{xr}.\\
&\qquad \gamma = \gamma_{xr} \to \mathsf{repr}\ \gamma_{xr} \to \mathsf{string}\\
&\quad \wedge\ \mathsf{int} = \mathsf{int}\\
&\quad \wedge\ \forall\beta\alpha.\exists\gamma_1\gamma_2.\beta + \alpha = \gamma_1 + \gamma_2\\
&\quad \wedge\ \forall\beta\alpha.\exists\gamma_1\gamma_2.\beta \times \alpha = \gamma_1 \times \gamma_2\\
&].\gamma\\
&\ \mathsf{in}\ \mathsf{true}
\end{aligned}
$$

At this point, there is no more implications in this constraint. The standard solver gives us the following solved form:

$$\mathsf{let}\ f : \forall\gamma_{xr}.\ \gamma_{xr} \to \mathsf{repr}\ \gamma_{xr} \to \mathsf{string}\ \mathsf{in}\ \mathsf{true}$$

Another path

$$
\begin{array}{l}
\mathsf{let}\ f : \forall\gamma.[\\
\exists\gamma_x\gamma_r. \\
\qquad \gamma = \gamma_x \to \mathsf{repr}\ \gamma_r \to \mathsf{string} \\
\quad \wedge\ \gamma_r = \mathsf{int} \Rightarrow \gamma_x = \mathsf{int} \\
\quad \wedge\ \forall\beta\alpha.\gamma_r = \alpha + \beta \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 + \gamma_2 \\
\quad \wedge\ \forall\beta\alpha.\gamma_r = \alpha \times \beta \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 \times \gamma_2 \\
].\gamma \\
\ \mathsf{in}\ \mathsf{true}
\end{array}
$$

Another tempting path to get rid of the first implication would be not to use the assumption by unifying γ_x and int.

Yann Régis-Gianas
yrg@irif.fr

Another path

$$
\begin{array}{l}
\text{let } f : \forall\gamma.[\\
\exists\gamma_x\gamma_r. \\
\qquad \gamma = \gamma_x \to \text{repr } \gamma_r \to \text{string} \\
\quad \wedge\ \gamma_x = \text{int} \\
\quad \wedge\ \forall\beta\alpha.\gamma_r = \alpha + \beta \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 + \gamma_2 \\
\quad \wedge\ \forall\beta\alpha.\gamma_r = \alpha \times \beta \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 \times \gamma_2].\gamma \\
\text{in true}
\end{array}
$$

This choice would force the solver to find an assignment of γ_r that makes the left hand side of the remaining implications unsatisfiable. There are many choices.

Yann Régis-Gianas
yrg@irif.fr

Another path

$$
\begin{array}{l}
\mathsf{let}\ f : \forall\gamma.[\\
\exists\gamma_x\gamma_r. \\
\qquad \gamma = \gamma_x \to \mathsf{repr}\ \gamma_r \to \mathsf{string} \\
\quad \wedge\ \gamma_x = \mathsf{int} \wedge\ \gamma_r = \mathsf{int} \\
\quad \wedge\ \forall\beta\alpha.\mathsf{false} \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 + \gamma_2 \\
\quad \wedge\ \forall\beta\alpha.\mathsf{false} \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 \times \gamma_2].\gamma \\
\mathsf{in}\ \mathsf{true}
\end{array}
$$

If we choose $\gamma_r = \mathsf{int}$, we get:

$$\mathsf{let}\ f : \mathsf{int} \to \mathsf{repr}\ \mathsf{int} \to \mathsf{string}\ \mathsf{in}\ \mathsf{true}$$

which is an instance of the first type scheme, which means that the simplification that was applied on the first implication prevented us from finding a more general type scheme.

Yann Régis-Gianas
yrg@irif.fr

Another path

$$
\begin{array}{l}
\mathsf{let}\ f : \forall\gamma.[\\
\exists\gamma_x\gamma_r. \\
\qquad \gamma = \gamma_x \to \mathsf{repr}\ \gamma_r \to \mathsf{string} \\
\quad \wedge\ \gamma_x = \mathsf{int} \wedge\ \gamma_r = \mathsf{string} \\
\quad \wedge\ \forall\beta\alpha.\mathsf{false} \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 + \gamma_2 \\
\quad \wedge\ \forall\beta\alpha.\mathsf{false} \Rightarrow \exists\gamma_1\gamma_2.\gamma_x = \gamma_1 \times \gamma_2].\gamma \\
\mathsf{in}\ \mathsf{true}
\end{array}
$$

If we choose $\gamma_r = \mathsf{string}$, we get:

$$\mathsf{let}\ f : \mathsf{int} \to \mathsf{repr}\ \mathsf{string} \to \mathsf{string}\ \mathsf{in}\ \mathsf{true}$$

which is not an instance of the first type scheme. Yet, it is not a very interesting specification for this function because there is no inhabitant in the type `repr string`.

Yann Régis-Gianas
yrg@irif.fr

More liberal solved forms

The loss of principal type schemes is very uncomfortable because a solver is forced to make some choices that may not be the same as the programmer's.

There is a way to recover principal type schemes. The idea is to allow constrained type schemes to be assigned to expressions as a valid result of type inference. This requires an extension of the type system to handle constrained type scheme, HMG(X) is such an extension [SP05].

In that case, we have both a correct and complete (decidable) type inference:

Theorem (Soundness [SP05])
$[\![e : \tau]\!], \Gamma \vdash e : \tau$ holds.

Theorem (Completeness [SP05])
If $C, \Gamma \vdash e : \forall \overline{\alpha}[D].\tau$ then $C \models \forall \overline{\alpha}.\ D \Rightarrow \mathit{def}\ \Gamma\ \mathit{in}\ [\![e : \tau]\!]$

Yann Régis-Gianas
yrg@irif.fr

Example

```
1  let f =
2    fun x r ->
3      match r with
4      | TyInt -> string_of_int x
5      | TySum (ty1, ty2) ->
6        (match x with Left x1 -> "L" | Right x2 -> "R")
7      | TyProd (ty1, ty2) ->  "P"
```

admits a most general constrained type scheme which is:

$$
\begin{aligned}
&\forall \alpha\beta[\\
&\qquad \alpha = \mathsf{int} \Rightarrow \beta = \mathsf{int} \\
&\quad \wedge \forall \beta_1\beta_2.\ \alpha = \beta_1 + \beta_2 \Rightarrow \exists \gamma_1\gamma_2.\ \beta = \gamma_1 + \gamma_2 \\
&\quad \wedge \forall \beta_1\beta_2.\ \alpha = \beta_1 \times \beta_2 \Rightarrow \exists \gamma_1\gamma_2.\ \beta = \gamma_1 \times \gamma_2 \\
&]. \ \beta \to \mathsf{repr}\ \alpha \to \mathsf{string}
\end{aligned}
$$

Yann Régis-Gianas
yrg@irif.fr

Discussion about constrained type schemes

Constrained type schemes are very expressive. However, there are several arguments against them.

First, by unleashing all the complexity of typing constraint inside type schemes, the complexity of the language of types is highly increased. As a consequence, **it may be hard for a programmer to understand what an inferred constrained type scheme means**, and, in particular, how to use a function with that type. The mind process to reason about the typeability of her program moves from simple unification-based problem to a complex satisfiability problem.

Second, the inferred constrained type scheme follows the same structure as the input program. Therefore, **constrained type schemes may show too many implementation details**, and, thus can break abstraction. As a consequence, such a specification is fragile, i.e. sensitive to minor modifications of programs.

Yann Régis-Gianas
yrg@irif.fr

Untractable type inference

Third, even if constraint solving is decidable, it is **not tractable**. Indeed, the lower-bound for constraint solving for the first-order theory of first-order term equality is **non-elementary** (DTIME($2^{2^{\cdots^{2^n}}}$)).

Despite of this theoretical result, one can hope that a complete solver could be engineered such that "practical cases" would be efficiently handled. This is an open problem, that has only been addressed using incomplete solvers in the literature. Anyway, if such a solver existed, giving a formal (and understandable) definition of the "practical cases" to the programmer would be a challenge too.

Should we give up?

A class of easy-to-infer programs

Let us be optimistic: there are a lot of programs that are easy to handle. Imagine that the type inference is given the following annotated version of `f`:

```
1  let f : 'a. 'a -> 'a repr -> string =
2    fun x r ->
3      match r with
4      | TyInt -> string_of_int x
5      | TySum (ty1, ty2) ->  (match x with Left x1 -> "L" | Right x2 -> "R")
6      | TyProd (ty1, ty2) ->  "P"
```

It seems straightforward to **check** that this function admits the provided type scheme. Why?

Yann Régis-Gianas
yrg@irif.fr

A constraint generation rule for user-annotated functions

By combining constraint generation rules for pattern matchings, functions and for user type annotations, we get an adhoc rule for user-annotated functions:

$$
\begin{array}{l}
[\![\mathsf{let}\ f : \forall\overline{\alpha}.\ \tau_1 \to \ldots \to \tau_n = \\
\quad \lambda x_1 \ldots x_n.\mathsf{match}\ x_i\ \mathsf{with}\ \overline{b} \\
\mathsf{in} \\
\quad T_2 \\
: \tau]\!] = \\
\quad \mathsf{let}\ f : \\
\quad \forall\underline{\overline{\alpha}}[\mathsf{def}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ \mathsf{in} \\
\qquad\qquad [\![\overline{b} : \tau_i \to \tau_n]\!] \\
\quad].\tau_1 \to \ldots \to \tau_n \\
\quad \mathsf{in} \\
\qquad [\![T_2 : \tau]\!]
\end{array}
$$

To adequately express the user type annotation, we underlined the type variable α to stress the requirement that this type variable

Yann Régis-Gianas
yrg@irif.fr

A class of easy-to-solve constraints

Our adhoc constraint generation rule for user-annotated function would produce:

$$
\begin{array}{l}
\mathsf{let}\ f : \forall \underline{\alpha}.\ [\\
\qquad \alpha = \mathsf{int} \Rightarrow \alpha = \mathsf{int} \\
\quad \wedge\ \forall \beta_1 \beta_2 . \alpha = \beta_1 + \beta_2 \Rightarrow \exists \gamma_1 \gamma_2 . \alpha = \gamma_1 + \gamma_2 \\
\quad \wedge\ \forall \beta_1 \beta_2 . \alpha = \beta_1 + \beta_2 \Rightarrow \exists \gamma_1 \gamma_2 . \alpha = \gamma_1 + \gamma_2 \\
]. \ \alpha \to \mathsf{repr}\ \alpha \to \mathsf{string} \\
\ \mathsf{in}\ \mathsf{true}
\end{array}
$$

All the type variables that appear in types on the left of implication connectives are universally quantified. As a consequence, we can solve these **rigid** implications "$C \Rightarrow D$" by trying to solve C first, and then, handle the two following cases:

- If $C \equiv \mathsf{false}$, we are done.
- If there is a most general unifier for C, apply it to D and try to solve D.

Yann Régis-Gianas
yrg@irif.fr

When does this approach break?

Let us transform this program using an $\eta-$expansion:

```
1  let f : 'a. 'a -> 'a repr -> string
2    fun y -> (fun x r ->
3      match r with
4      | TyInt -> string_of_int x
5      | TySum (ty1, ty2) ->
6        (match x with Left x1 -> "L" | Right x2 -> "R")
7      | TyProd (ty1, ty2) ->  "P") y
```

Our adhoc rule cannot be applied. Thus, the typing constraint does not contain rigid implications anymore. Let us look at this typing constraint though.

Yann Régis-Gianas
yrg@irif.fr

Let us try harder

$$
\begin{aligned}
&\text{let } f : \forall \underline{\alpha}.[\exists \gamma_x.\\
&\quad \alpha = \text{int} \Rightarrow \gamma_x = \text{int}\\
&\quad \wedge\, \forall \beta_1 \beta_2 . \alpha = \beta_1 + \beta_2 \Rightarrow \exists \gamma_1 \gamma_2 . \gamma_x = \gamma_1 + \gamma_2\\
&\quad \wedge\, \forall \beta_1 \beta_2 . \alpha = \beta_1 \times \beta_2 \Rightarrow \exists \gamma_1 \gamma_2 . \gamma_x = \gamma_1 \times \gamma_2\\
&\quad \wedge\, \gamma_x = \alpha\\
&].\alpha \to \text{repr } \alpha \to \text{string}\\
&\text{in true}
\end{aligned}
$$

If we first solve simple unification constraints, and we **suspend** implications that are not rigid, we get:

$$
\begin{aligned}
&\text{let } f : \forall \underline{\alpha}.[\\
&\quad \alpha = \text{int} \Rightarrow \alpha = \text{int}\\
&\quad \wedge\, \forall \beta_1 \beta_2 . \alpha = \beta_1 + \beta_2 \Rightarrow \exists \gamma_1 \gamma_2 . \alpha = \gamma_1 + \gamma_2\\
&\quad \wedge\, \forall \beta_1 \beta_2 . \alpha = \beta_1 \times \beta_2 \Rightarrow \exists \gamma_1 \gamma_2 . \alpha = \gamma_1 \times \gamma_2\\
&].\alpha \to \text{repr } \alpha \to \text{string}\\
&\text{in true}
\end{aligned}
$$

Eventually, the rewritten constraint only contains rigid implications.

How long can the typing constraints be suspended?

To avoid complex constrained type schemes to arise, we must interleaved constraint generation and constraint solving at each `let` in order to force the type inference engine to ascribe an ML type scheme to the `let`-bound variable.

Yann Régis-Gianas
yrg@irif.fr

Another example

Consider now the following example [Sch+09]:

```
1  type t 'a =
2    | K1 : int -> bool t
3    | K2 : forall 'a. 'a list -> 'a t
4
5  let f1 = function
6    | K1 n -> n > 0
7
8  let f2 = function
9    | K1 n -> n > 0
10   | K2 xs -> is_nil xs
```

What would you do if you were a type inference engine?
(Assume that `is_nil : 'a list -> bool`.)

Yann Régis-Gianas
yrg@irif.fr

The typing constraint of `f1`

$$
\begin{aligned}
&\text{let } f1 : \forall\alpha[\\
&\quad \exists\gamma_1\gamma_2.\alpha = \gamma_1 \to \gamma_2 \\
&\quad \wedge\, \exists\gamma_3.\ \gamma_1 = t\ \gamma_3 \\
&\quad \wedge\, \forall\gamma_4.\ (\gamma_1 = t\ \gamma_4\ \wedge\ \gamma_4 = \text{bool} \Rightarrow \gamma_2 = \text{bool}) \\
&]\alpha \text{ in } \ldots
\end{aligned}
$$

First, we simplify non implication constraints.

Yann Régis-Gianas
yrg@irif.fr

The typing constraint of `f1`

$$\begin{array}{l}\mathsf{let}\ f1 : \forall\gamma_2\gamma_3[\\ \quad \gamma_3 = \mathsf{bool} \Rightarrow \gamma_2 = \mathsf{bool})\\ ].\mathsf{t}\ \gamma_3 \to \gamma_2\ \mathsf{in}\ \ldots\end{array}$$

The following two incompatible assignments satisfy the implication:

- $\gamma_2 \mapsto \mathsf{bool}$, yielding $f1 : \forall\alpha.\mathsf{t}\ \alpha \to \mathsf{bool}$
- $\gamma_2 \mapsto \gamma_3$, yielding $f1 : \forall\alpha.\mathsf{t}\ \alpha \to \alpha$

These are two most-general type schemes that are incomparable. A type inference engine should **reject** that program and should ask the programmer to make a choice by putting a type annotation in front of `f1`.

Typing constraint of `f2`

$$
\begin{aligned}
&\text{let } f2 : \forall\alpha[\\
&\quad \exists\gamma_1\gamma_2.\alpha = \gamma_1 \to \gamma_2 \\
&\quad \wedge\ \exists\gamma_3.\ \gamma_1 = t\ \gamma_3 \\
&\quad \wedge\ \forall\gamma_4.\ (\gamma_1 = t\ \gamma_4\ \wedge\ \gamma_4 = \mathsf{bool} \Rightarrow \gamma_2 = \mathsf{bool}) \\
&\quad \wedge\ \forall\gamma_5.\ (\gamma_1 = t\ \gamma_5 \Rightarrow \gamma_2 = \mathsf{bool}) \\
&]. \alpha \text{ in } \ldots
\end{aligned}
$$

Again, we first simplify non implication constraints.

Yann Régis-Gianas
yrg@irif.fr

Typing constraint of `f2`

$$\begin{array}{l}\mathsf{let}\ f2 : \forall\gamma_2\gamma_3[\\ \quad \gamma_3 = \mathsf{bool} \Rightarrow \gamma_2 = \mathsf{bool})\\ \quad \wedge\ \gamma_2 = \mathsf{bool})\\ ].\ \mathsf{t}\ \gamma_3 \to \gamma_2\ \mathsf{in}\ \ldots\end{array}$$

At some point, we have simplified the second implication, leading to an unconditional assignment of γ_2. This time the information about γ_2 comes from the **outside** of the first clause. As a result, the corresponding substitution can be safely applied to the right hand side of the implication.

Yann Régis-Gianas
yrg@irif.fr

Typing constraint of `f2`

$$\begin{array}{l}\mathsf{let}\ f2 : \forall\gamma_2\gamma_3[\\ \quad \gamma_3 = \mathsf{bool} \Rightarrow \mathsf{bool} = \mathsf{bool})\\ ].\ \mathsf{t}\ \gamma_3 \to \mathsf{bool}\ \mathsf{in}\ \ldots\end{array}$$

The implication can be removed because its right hand side is equivalent to **true** and we finally obtain:

$$f2 : \forall\alpha.\mathsf{t}\ \alpha \to \mathsf{bool}$$

Notice that the solver has not made any choice.

Can a solver make a distinction between these two cases?

The idea of the `OutsideIn` approach [Sch+09] is to forbid unification of type variables if it could be influenced by a type refinement. We **mark** these unification variables as **untouchable**. The only typing constraints that help learning something about these variables are not themselves constrained by local assumptions.

For the first function, we have:

$$\begin{array}{l}\mathsf{let}\ f1 : \forall\alpha[\\ \quad \exists\gamma_1\gamma_2.\alpha = \gamma_1 \to \gamma_2 \\ \quad \wedge\ \exists\gamma_3.\ \gamma_1 = t\ \gamma_3 \\ \quad \wedge\ [\gamma_2\ \gamma_1](\forall\gamma_4.\ (\gamma_1 = t\ \gamma_4\ \wedge\ \gamma_4 = \mathsf{bool}) \Rightarrow \gamma_2 = \mathsf{bool}) \\ ].\alpha\ \mathsf{in}\ \ldots\end{array}$$

"$[\overline{\gamma}](\forall\overline{\beta}.C_1 \Rightarrow C_2)$" is an implication constraint with the untouchables $\overline{\gamma}$.

Yann Régis-Gianas
yrg@irif.fr

Simplifying implication constraints

To simplify a constraint of the form $[\overline{\gamma}](\forall\overline{\beta}.C_1 \Rightarrow C_2)$, here is how to proceed:

1. Simplify C_1 and get a solved form. It encodes an mgu ϕ, whose domain may contain some untouchables.
2. Apply ϕ to C_2 and simplify C_2.
3. Check that the resulting mgu ϕ' **does not assign** one of the untouchables.

Yann Régis-Gianas
yrg@irif.fr

Example

$$\begin{array}{l}\mathsf{let}\ f1 : \forall\gamma_2\gamma_3[\\ \quad [\gamma_3\ \gamma_2](\gamma_3 = \mathsf{bool} \Rightarrow \gamma_2 = \mathsf{bool})\\ ].\mathsf{t}\ \gamma_3 \to \gamma_2\ \mathsf{in}\ \ldots\end{array}$$

The mgu $\gamma_3 \mapsto \mathsf{bool}$ is applied to $\gamma_2 = \mathsf{bool}$ with no effect. As $\gamma_2 \mapsto \mathsf{bool}$ assigns the untouchable γ_2, this constraint is rejected.

Yann Régis-Gianas
yrg@irif.fr

Example

In the second case, the typing constraint:

$$\begin{array}{l}\mathsf{let}\ f2 : \forall\gamma_2\gamma_3[\\ \quad [\gamma_3\ \gamma_2]\gamma_3 = \mathsf{bool} \Rightarrow \gamma_2 = \mathsf{bool})\\ \quad \wedge\ \gamma_2 = \mathsf{bool})\\ ].\ \mathsf{t}\ \gamma_3 \rightarrow \gamma_2\ \mathsf{in}\ \ldots\end{array}$$

is simplified into

$$\begin{array}{l}\mathsf{let}\ f2 : \forall\gamma_2\gamma_3[\\ \quad [\gamma_3\ \gamma_2]\gamma_3 = \mathsf{bool} \Rightarrow \mathsf{bool} = \mathsf{bool})\\ ].\ \mathsf{t}\ \gamma_3 \rightarrow \mathsf{bool}\ \mathsf{in}\ \ldots\end{array}$$

which will not be rejected because $(\mathsf{bool} = \mathsf{bool}) \equiv \mathsf{true}$.

Metatheorems

This solver is correct and the good news is that it always returns unquestionable answers:

Theorem (Principality)

If a type scheme is inferred, it is a principal type scheme.

Besides, it is a conservative extension of ML ([**xi03**] was not):

Theorem (Conservative extension of ML)

If a term T has principal type σ in ML without GADTs, then the same type scheme will be inferred by the *OutsideIn* algorithm.

Yann Régis-Gianas
yrg@irif.fr

What about completeness?

Unfortunately, without any surprise, there are some programs that do enjoy principal types but that are rejected by the `OutsideIn` system. For instance:

```
1  type 'a t = K : int t
2  (* Function f admits the following principal type scheme: *)
3  let f : forall 'a. 'a t -> string =
4    fun x -> let h = match x with K -> 42 in
5      "foobar"
```

The typing constraint looks like:

$$\begin{array}{l}\mathsf{let}\ f : \forall\gamma_1\gamma_2[\\ \quad \mathsf{let}\ h : \forall\gamma_3[[\gamma_1\ \gamma_3](\gamma_1 = \mathsf{int} \Rightarrow \gamma_3 = \mathsf{int})].\gamma_3\ \mathsf{in}\\ \qquad \gamma_2 = \mathsf{string}\\ ].\ \gamma_1 \to \gamma_2\end{array}$$

The implication $[\gamma_1\ \gamma_3](\gamma_1 = \mathsf{int} \Rightarrow \gamma_3 = \mathsf{int})$ will be rejected by the solver although the assignment $\gamma_3 \mapsto \mathsf{int}$ is fine in that example.

A declarative definition of completeness for `OutsideIn`

The inference engine **is not complete** with respect to the canonical extension of ML with GADTs.

This is damageable for the **predictability** of the system. What kind of mental image of the type system should the programmer use while programming in order to decide the **amount of type annotation** that is required to please the type inference engine?

A declarative definition of completeness for `OutsideIn`

A possible answer would be to give up this idea and suggest a "Try and Fix" usage of the type inference engine. Yet, this lack of specification for accepted programs has two pitfalls.

First, it is not guaranteed that the set of accepted programs will be preserved in future versions of the type checker because **this set relies on the internals of the constraint solver**.

Second, **how could a programmer discriminate rejected programs** between ill-typed and "not annotated enough" programs?

Yann Régis-Gianas
yrg@irif.fr

A declarative definition of completeness for `OutsideIn`

The authors of the OutsideIn approach designed a declarative type system for which a completeness metatheorem holds. They hope that type system to act as a documentation.

Yann Régis-Gianas
yrg@irif.fr

Explaining accepted programs to the lambda programmer

"First, imagine that your pattern matching is replaced by a black-box (a call to an unknown function for instance), has your expression (and its variables) the type you want? Second, open the black-box, are you able to type-check the pattern matching without backtracking on your initial assumptions? If the answer is yes, your expression has the type you need."

Yann Régis-Gianas
yrg@irif.fr

Summary

The **`OutsideIn`** approach deals with the interrelated problems:

1. What are the types of the sub-expressions?
2. What are the available type equalities?
3. How are the type equalities used?
4. Is a branch dead?

by removing the dependency of the first problem with respect to the others. Indeed, as the type equalities cannot influence the inferred types of expressions that host pattern matchings over GADTs, they can be computed first and, then, the other problems are tackled to infer the types inside the pattern matching.

Another design choice

Even if the previous declarative type system is relatively easy to explain, it was designed a posteriori to fit the class of programs handled by the type inference engine. As a consequence, it is not that close to the (declarative) canonical type system.

There is another piece of work [PR06] that takes an opposite design method:

- first, define a type system with principal types that is as close as possible to the canonical one such that it could be used as a replacement ;
- second, devise an algorithm that reconstruct a typing derivation of the former system from a term of the latter.

Plan

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

GADTs encodings

Type inference in ML extended with GADTs

Why is type inference difficult?

Stratified type inference

`OutSideIn(X)`

Yann Régis-Gianas
yrg@irif.fr

MLGX

Coming back to the interrelated problems:

1. What are the types of the sub-expressions?
2. What are the available type equalities?
3. How are the type equalities used?
4. Is a branch dead?

...the **stratified approach** removes the problems 1 and 2 through the introduction of a programming language called MLGX that makes an **explicit** usage of type equalities.

Thus, there is no more conversion rule in MLGX.

Yann Régis-Gianas
yrg@irif.fr

Restriction 1 : Scrutinee must be annotated

In MLGX, **all pattern matchings must be annotated**.

Only this user type annotation will be used to determine the available type equalities. For instance, in MLGX, if one has written:

```
1  let f : forall 'a. 'a -> 'a repr -> string =
2    fun x r ->
3      match r with
4      | TyInt -> ...
5      | TySum (ty1, ty2) -> ...
6      | TyProd (ty1, ty2) ->  ...
```

...there is no available type equality in the body of the branches.

Yann Régis-Gianas
yrg@irif.fr

Restriction 1 : Scrutinee must be annotated

In MLGX, **all pattern matchings must be annotated**.

Only this user type annotation will be used to determine the available type equalities. For instance, in MLGX, if one has written:

```
1  let f : forall 'a. 'a -> 'a repr -> string =
2    fun x r ->
3      match (r : 'a repr) with
4      | TyInt -> (* α = int *) ...
5      | TySum (ty1, ty2) -> (* ∃β1 β2.α = β1 + β2*) ...
6      | TyProd (ty1, ty2) ->  (* ∃β1 β2.α = β1 × β2*) ...
```

...there are type equalities in the body of the branches.

A complete syntax to write all kind of type annotations

It can be useful to allow **partial** user type annotations to be written, that is why we introduce new syntactic classes for type annotations:

$$\begin{array}{lcl} \theta & ::= & \exists \overline{\gamma}.\tau \\ \zeta & ::= & \exists \overline{\gamma}.\sigma \end{array}$$

In a type annotation θ (resp. a type scheme annotation ζ), the type variables $\overline{\gamma}$ bind inside τ (resp. σ). As flexible type variables, they represent unknown types.

We also need to name the local type variables that are introduced by patterns in the body of the branches in order to use them inside type annnotations, that is why we extend the syntax of patterns to:

$$p \quad ::= \quad K\overline{\beta}\,(x_1, \ldots, x_n)$$

Yann Régis-Gianas
yrg@irif.fr

Type checking rules for pattern matching

X-Match

$$\frac{E, \Gamma \vdash (t : \theta) : \tau_1 \qquad \forall i, E, \Gamma \vdash (p_i : \theta) \Rightarrow t_i : \tau_1 \to \tau_2}{E, \Gamma \vdash \mathsf{match}\ (t : \theta)\ \mathsf{with}\ p_1 \Rightarrow t_1 \mid \ldots \mid p_n \Rightarrow t_n : \tau_2}$$

X-Branch

$$\frac{p : \varepsilon\ \overline{\tau}_1\ \overline{\tau}'_2 \vdash (\overline{\beta}, E', \Gamma') \qquad E \wedge E', \Gamma\Gamma' \vdash t : \tau}{E, \Gamma \vdash (p : \exists \overline{\gamma}.\varepsilon\ \_\ \overline{\tau}'_2) \Rightarrow t : \varepsilon\ \overline{\tau}_1\ \_ \to \tau}$$

$$\text{with} \begin{cases} \overline{\beta} \# \mathrm{FTV}(\Gamma, E, \tau) \\ \overline{\gamma} \# \mathrm{FTV}(\Gamma, E, \tau, t) \end{cases}$$

The generalized type parameters $\overline{\tau}'_2$, which are the source of type equalities, are extracted from the user type annotation ascribed to the scrutinee. As a result, E can be deduced everywhere from the type annotations of the program only. Notice that the user type annotation is checked.

Example

```
1  type ('a, 'n) list =
2  | Nil : ('a, zero) list
3  | Cons : forall 'a n. 'a * ('a, 'n) list -> ('a, 'n succ) list
4
5  let head = forall n.
6    match (l : exists 'a. ('a, n succ) list) with Cons (x, _) -> x
```


Yann Régis-Gianas
yrg@irif.fr

Restriction 2 : Coercions must be explicited

In MLGX, **the usage of type equalities are explicited** using **coercions** κ :

$$\kappa ::= \exists \overline{\gamma}.(\tau_1 \triangleright \tau_2)$$

where $\overline{\gamma}$ binds into τ_1 and τ_2.

The rule for coercion replaces the implicit conversion by allowing a term of type τ_1 to be seen as a term of type τ_2 only if it has been explicitly required and if the two types are equal under the current assumptions:

$$\frac{E, \Gamma \vdash t : \tau_1 \qquad \kappa \preceq \tau_1 \triangleright \tau_2 \qquad E \models \kappa}{E, \Gamma \vdash (t : \kappa) : \tau_2}$$

Notice that if E is provided, the annotation κ can be checked orthogonally to t.

Yann Régis-Gianas
yrg@irif.fr

MLGI is an implicit version of MLGX

What is the link between MLGI and MLGX?

Let us define an equivalence of terms with respect to user type annotations:

$$\frac{t \equiv t' \qquad \overline{\alpha} \,\#\, \mathrm{FTV}(t)}{t \equiv \forall \overline{\alpha}.t'} \qquad \frac{t \equiv t'}{t \equiv (t' : \theta)} \qquad \frac{t \equiv t'}{t \equiv (t' : \kappa)}$$

Theorem (Completeness with assistance for MLGX)

If $E, \Gamma \vdash t : \sigma$ holds in MLGI, then there exists a term t' such that $t \equiv t'$ and $E, \Gamma \vdash t' : \sigma$ holds in MLGX.

Any typing derivation of MLGI can be represented in MLGX if enough type annotations are provided by the programmer.

Yann Régis-Gianas
yrg@irif.fr

Explaining MLGX to the lambda programmer

To explain MLGX to a programmer, we can focus on the features of GADTs:

1. If you want a type equality to be available, the scrutinee must be annotated. If the scrutinee is not annotated, there is no available type equality.
2. If you want a conversion to be applied, write it down using a coercion. If it is not written, it means that you do not want a conversion to happen.

There is no reference to a type inference algorithm here.

Yann Régis-Gianas
yrg@irif.fr

Type inference for MLGX

Type inference is easy in MLGX: it is a straightforward extension of Hindley-Milner type inference with user type annotations.

Indeed, as the local assumptions can be computed direcly from the user type annotations, **coercions can be checked in a first pass**.

Then, each coercion $(t : \exists\overline{\gamma}.\tau_1 \triangleright \tau_2)$ can be interpreted as the application to t of a function $\mathsf{cast}_{\tau_1 \triangleright \tau_2}$ of type $\forall\overline{\gamma}.\tau_1 \to \tau_2$, that computationally behaves like the identity.

Yann Régis-Gianas
yrg@irif.fr

More formally...

$$[\![E \vdash (t : \exists \overline{\gamma}.\ \tau_1 \triangleright \tau_2) : \tau]\!]$$
$$= \exists \overline{\gamma}.([\![E \vdash t : \tau_1]\!]) \wedge \tau_2 = \tau)$$

$$[\![E \vdash \mathsf{match}\ (t : \theta)\ \mathsf{with}\ p_1 \Rightarrow t_1 \mid \ldots \mid p_n \Rightarrow t_n : \tau]\!]$$
$$= \exists \gamma.([\![E \vdash (t : \theta) : \gamma]\!] \wedge \textstyle\bigwedge_i [\![E \vdash (p_i : \theta) \Rightarrow t_i : \gamma \to \tau]\!])$$

$$[\![E \vdash (K\overline{\beta}\,(x_1, \ldots, x_n) : \exists \overline{\gamma}.\varepsilon\ \_\ \overline{\tau}'_2) \Rightarrow t : \tau' \to \tau]\!]$$
$$= \exists \overline{\alpha}.(\exists \overline{\gamma}_2{}'.(\tau' = \varepsilon\ \overline{\alpha}\ \overline{\gamma}_2{}') \wedge$$
$$\forall \overline{\beta}\overline{\gamma}.\ \mathsf{def}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ \mathsf{in}$$
$$[\![E \wedge (\overline{\tau}'_2 = \overline{\tau}) \vdash t : \tau]\!])$$
$$\text{with } K :: \forall \overline{\alpha}\overline{\beta}.\ \tau_1 \times \ldots \tau_n \to \varepsilon\ \overline{\alpha}\ \overline{\tau}$$

Yann Régis-Gianas
yrg@irif.fr

Complete type inference for MLGX

Theorem (MLGX has principal types)

ϕ is a unifier for $[\![E \vdash t : \tau]\!]$ if and only if $E, \phi(\Gamma) \vdash t : \phi(\tau)$ holds in MLGX.

No more choices are done by the type inference engine because these are explicited by the presence or the lack of user type annotations in the term.

Yann Régis-Gianas
yrg@irif.fr

Example

Here is the function `f` in MLGX:

```
let f = forall 'a.
  fun x r ->
    match (r : 'a repr) with
    | TyInt -> string_of_int (x : 'a :> int)
    | TySum ['b1 'b2] (ty1, ty2) ->
        (match (x : 'a :> 'b1 + 'b2) with
         | Left x1 -> "L"
         | Right x2 -> "R")
    | TyProd ['b1 'b2] (ty1, ty2) ->  "P"
```

This is what the programmer has in mind when she wants to convince herself that this program is well-typed.

Yet, writing all the coercion is painful.

Yann Régis-Gianas
yrg@irif.fr

Heterogeneous difficulties

We would like to write the more concise toplevel annotation in and deduce automatically the other annotations.

```
let f : forall 'a. 'a -> 'a repr -> string =
  fun x  r ->
    match (r : 'a repr) with
    | TyInt -> string_of_int (x : 'a :> int)
    | TySum 'b1 'b2 (ty1, ty2) ->
        (match (x : 'a :> 'b1 + 'b2) with
         | Left x1 -> "L"
         | Right x2 -> "R")
    | TyProd 'b1 'b2 (ty1, ty2) ->  "P"
```

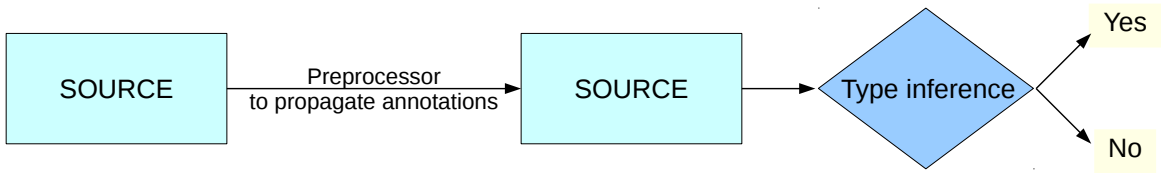
...because all the required typing information is already present in the red annotation and can be **propagated** to the places where the green annotations are.

Yann Régis-Gianas
yrg@irif.fr

Stratified type inference

MLGX is a robust, well understood, **back-end**, that serves as a final reference to decide which typing derivation is being targeted by the programmer.

We will design now an adhoc **front-end**, that works as a preprocessor to propagate type annotations and reconstruct an MLGX term with more annotations.



Yann Régis-Gianas
yrg@irif.fr

On the incompleteness of the front-end

This algorithm is clearly incomplete with respect to MLGI: starting with a term without any annotations, it will not be able to reconstruct the programmer's choices by magic. Yet, we want this propagation to be **sound** and as simple and as **predictable** as possible.

By **sound**, we mean that the inserted type annotations are correct with respect to the programmer's intent. In other words, the inserted type annotations should not make choices that were not already latent in the initial type annotations.

Being **predictable** is an informal property. We replace it with a **locality** condition, following previous work on **local type inference** [**pierce98**]:

> "Missing type annotations [should be] recovered using only information from adjacent nodes in the syntax tree, without long-distance constraints such as unification variables."

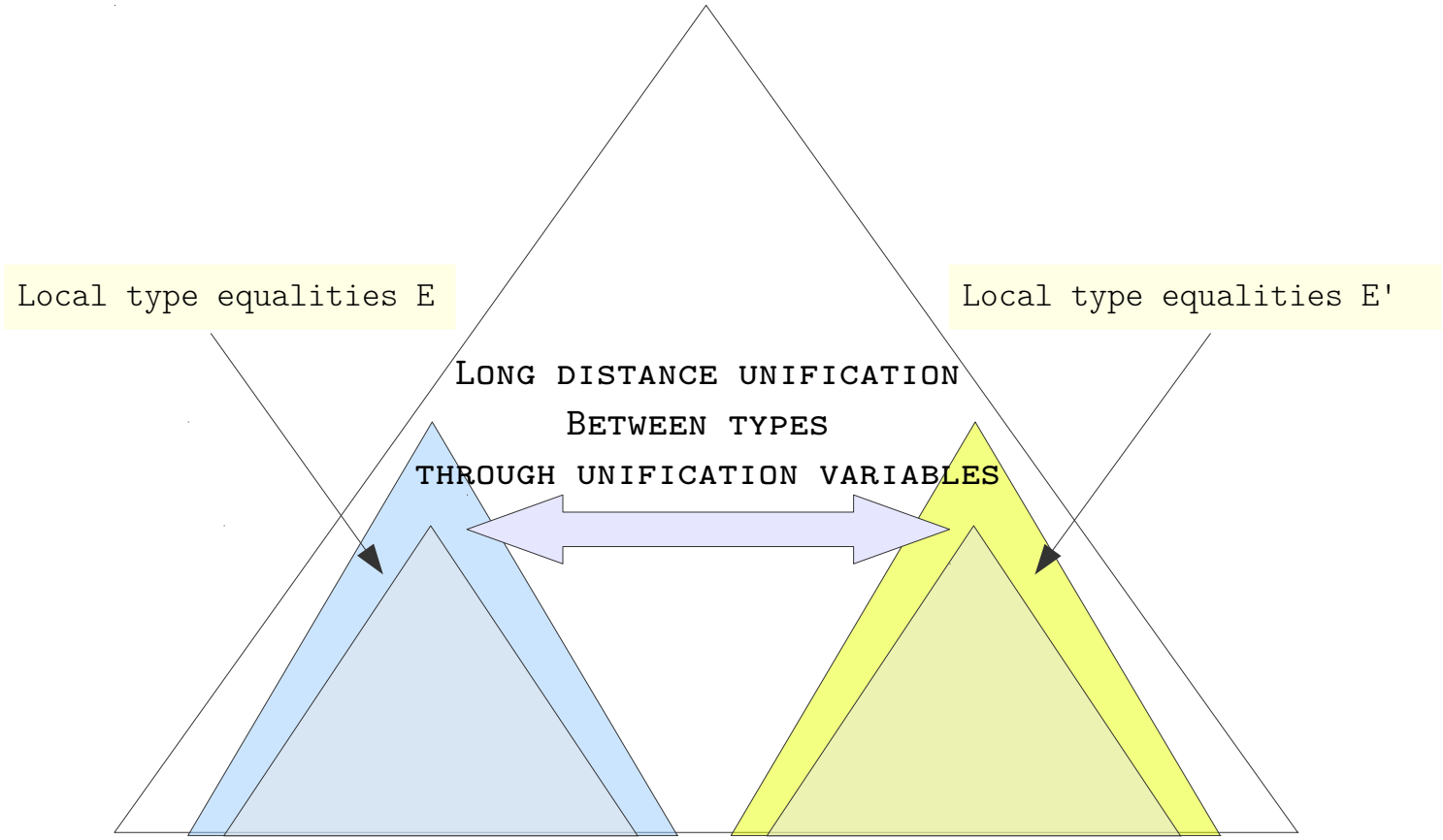
Yann Régis-Gianas
yrg@irif.fr

Picturing local type inference

ML type inference

Yann Régis-Gianas
yrg@irif.fr

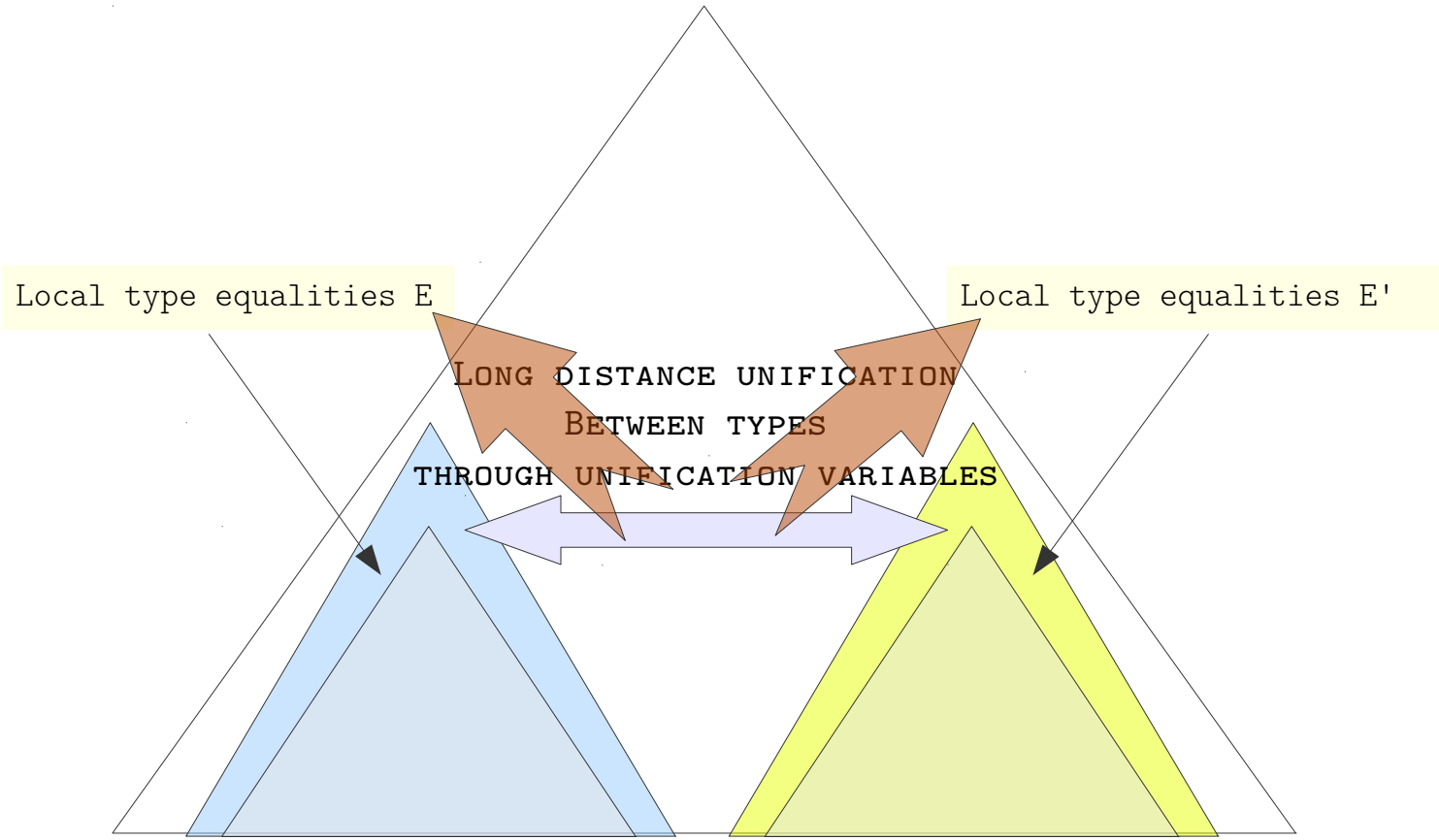
Picturing local type inference



ML type inference with GADTs needs unification modulo E and E'...

Yann Régis-Gianas
yrg@irif.fr

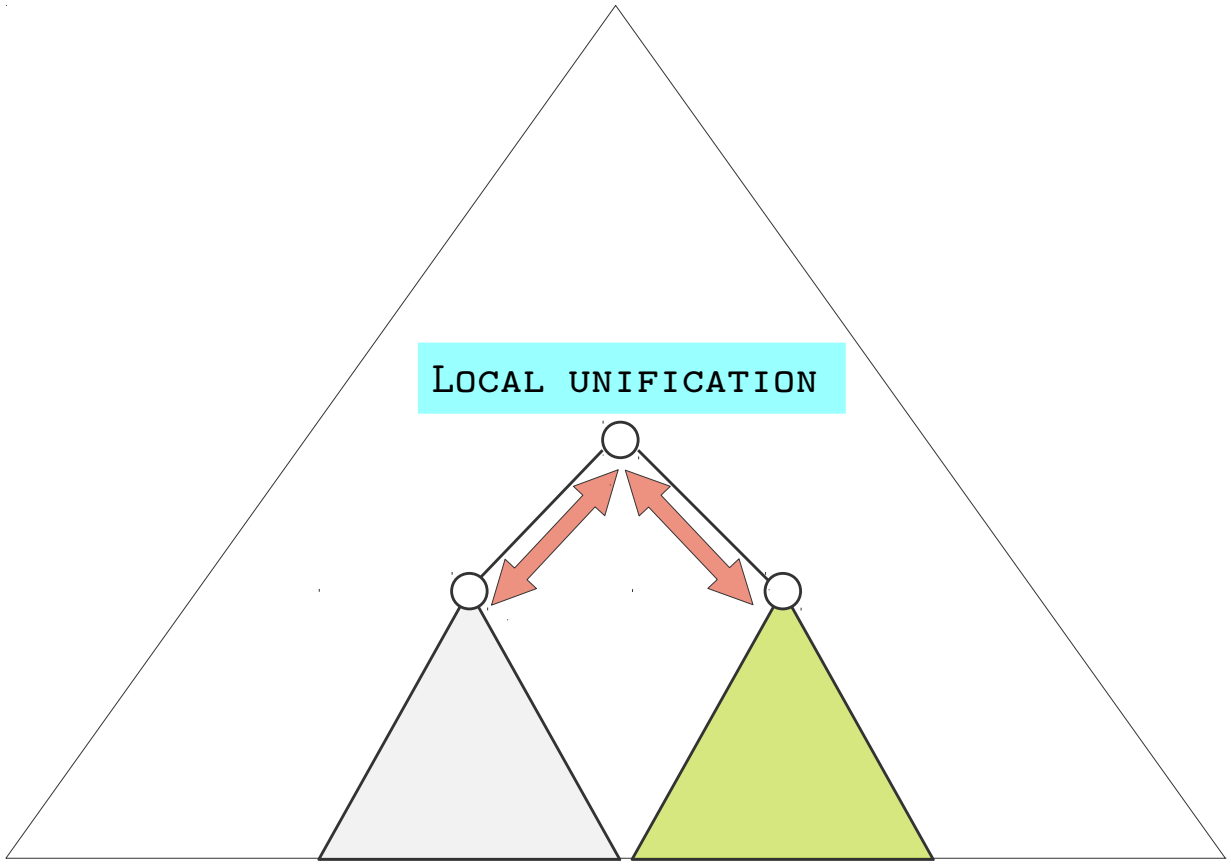
Picturing local type inference



GADTs need unification modulo E and E'..., which depend on unification!

Yann Régis-Gianas
yrg@irif.fr

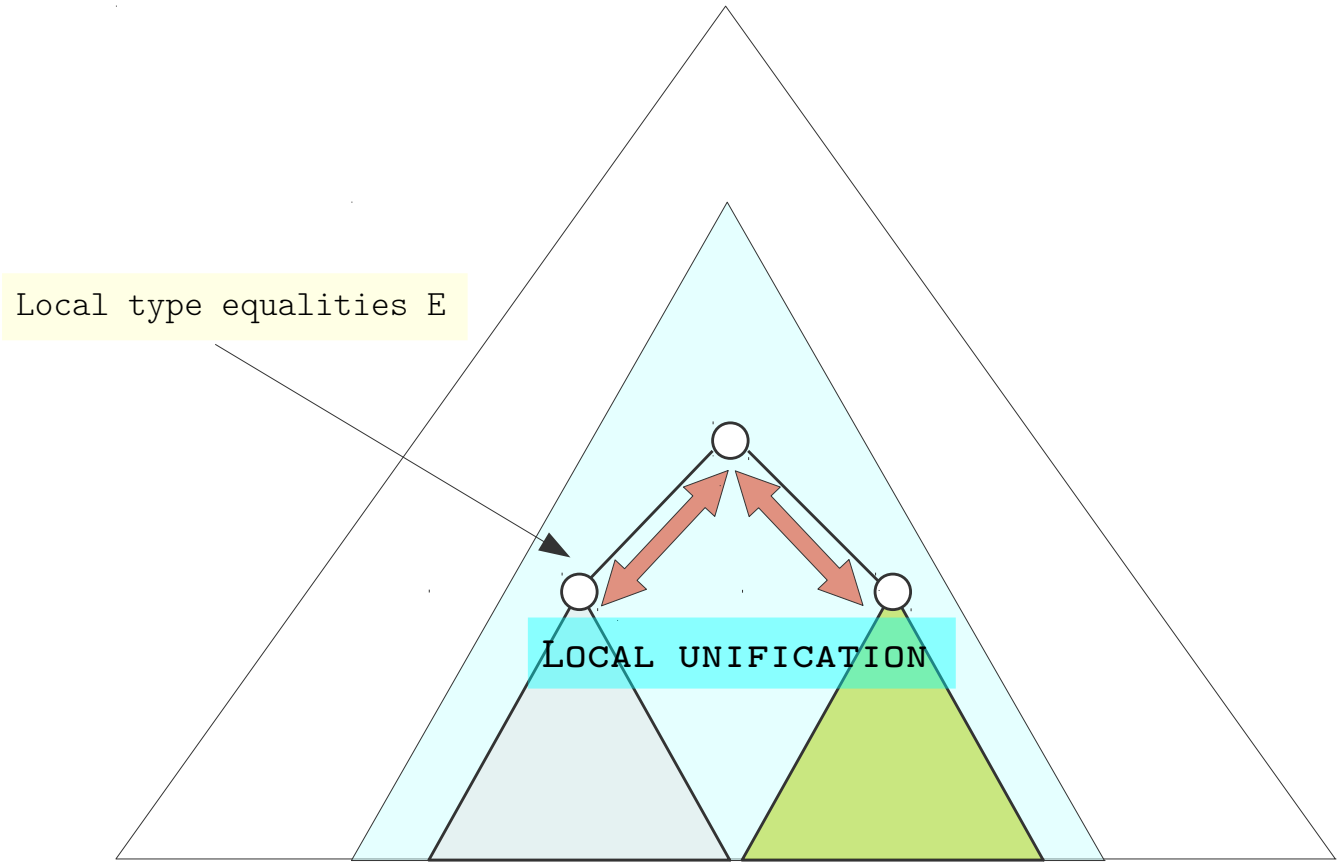
Picturing local type inference



Local type inference uses local unification.

Yann Régis-Gianas
yrg@irif.fr

Picturing local type inference



In presence of GADTs, we will use local unification modulo a known E.

Yann Régis-Gianas
yrg@irif.fr

Example

```
1  type 'a t =
2  | T1 : int -> int t
3  | T2 : 'a list -> 'a t
4
5  let apply = forall 'a.
6    fun (x : 'a) p (t : 'a repr) ->
7      match t with
8      | T1 n -> p x && n > 0
9      | T2 l -> (p : 'a -> bool) (List.hd xs)
```

The user annotation says that `x : 'a`. Thus, in the first branch, in expression "`p x`", the variable `p` is applied to an argument that have both types `'a` and `int`. To be correct, the propagation must not blindly unify the argument type of `p` outside the branch with `int` because otherwise this would contradict the annotation of the second branch.

Yann Régis-Gianas
yrg@irif.fr

Precautious propagations

Local assumptions define **equivalence classes** between types. At one particular moment, the algorithm that propagates the annotation may hold **a particular representant** τ of an equivalence class induced by a particular set of type equalities E.

As long as we stay under a fixed set of type equalities E, it is correct to unify type annotations modulo E.

On the contrary, it is incorrect to syntactically unify τ with an annotation that was written outside the pattern matching, where another set of type equalities holds.

Example

Take $E = (\alpha = \beta_1 \times \beta_2)$, it is fine to compute $\alpha \sqcup \gamma \times \beta_2$ under E. (We get $\beta_1 \times \beta_2$ or α.) Yet, if $\gamma \times \beta_2$ was written under $E = \mathsf{true}$, then it cannot be unified into α.

How to formalize that idea?

Yann Régis-Gianas
yrg@irif.fr

Plan

Informal presentation of Generalized Algebraic Data types

Formal presentation of GADTs

Applications

GADTs encodings

Type inference in ML extended with GADTs

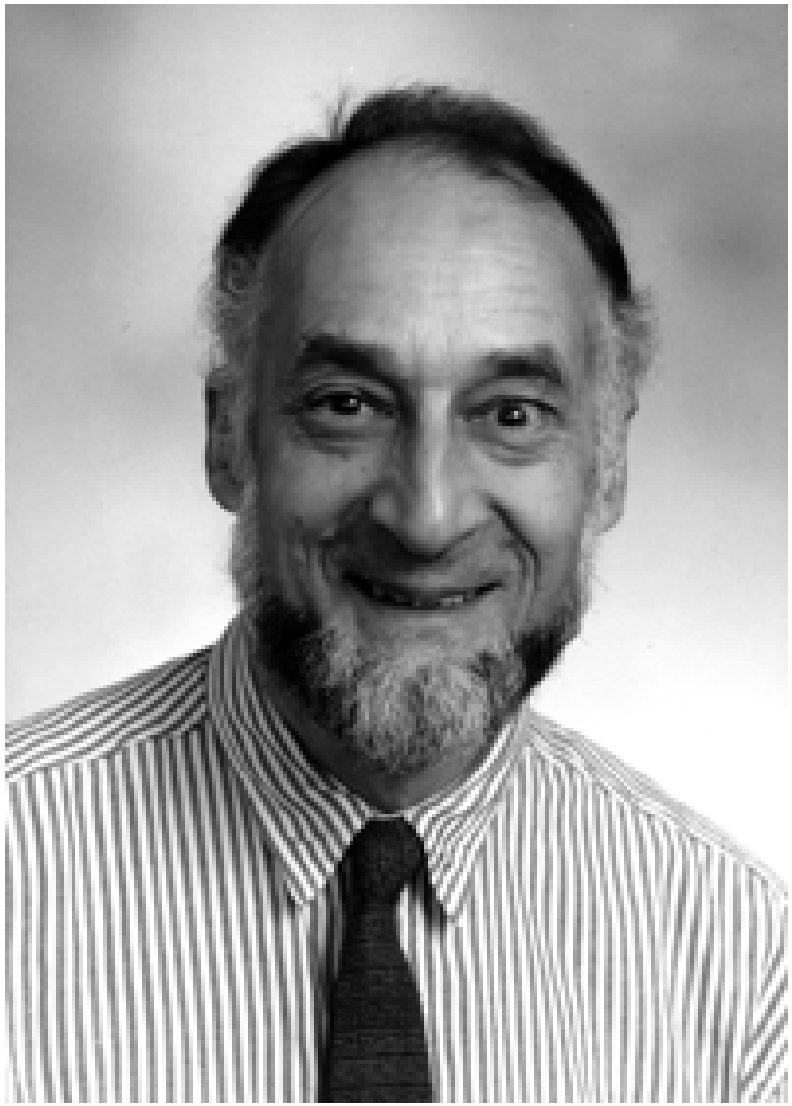
Why is type inference difficult?

Stratified type inference

`OutSideIn(X)`

Yann Régis-Gianas
yrg@irif.fr

Recreational break



Recreational break

Another design choice: Dropping let generalization

We will now present a third design choice that consists in getting rid of one complex central part of ML type inference which is generalization on **let** binding.

This (relevant?) simplification makes type inference in presence of GADTs easier while allowing the inference of principal constrained type schemes. This design is useful to handle both GADTs and type classes in the type inference algorithm.

Yann Régis-Gianas
yrg@irif.fr

Example (inspired by [Vyt+11])

```
let fr : forall 'a. 'a -> 'a repr -> bool =
  fun x r ->
    let g = fun z -> not x in
    match r with
    | TyBool -> g ()
    | _ -> true
```

In a system with generalization on let binding and where constrained type schemes can be inferred, this program is well-typed. Can you explain why?

Yann Régis-Gianas
yrg@irif.fr

Example (inspired by [Vyt+11])

```
let fr : forall 'a. 'a -> 'a repr -> bool =
  fun x r ->
    let g : forall 'b ['a = bool]. 'b -> bool = fun z -> not x in
    match r with
    | TyBool -> g ()
    | _ -> true
```

Returning a constrained type schemes for a function is always possible when a unification constraint is not satisfied locally but may be satisfied under some local assumptions at the call site.

Is that really what we want?

Yann Régis-Gianas
yrg@irif.fr

Yes! Because this validates let-expansion.

let introduces a local definition that can be morally inlined at each occurence of the bound variable. The following program is clearly well-typed, so should the version of this program with a folded local definition!

```
1  let fr : forall 'a. 'a -> 'a repr -> bool =
2    fun x r ->
3      match r with
4      | TyBool -> fun z -> not x
5      | _ -> true
```


Yann Régis-Gianas
yrg@irif.fr

No! This program should be rejected.

```
1  let fr : forall 'a. 'a -> 'a repr -> bool =
2    fun x r ->
3      let g = fun z -> not x in
4      match r with
5      | TyBool -> g ()
6      | _ -> true
```

If the programmer has not made explicit her typing local assumptions, this is probably because she is not aware of them. Thus, no local assumption should be assumed. (And this program should be rejected.)

Yann Régis-Gianas
yrg@irif.fr

Let us generalize the idea of non generalization

The typing rule for the unannotated `let` is now conformed to the syntactic sugar `let x = e1 in e2` $\equiv$ `(fun x -> e2) e1`:

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : \tau_2}$$

So, the typing constraint is simply the combination of the ones for application and lambda abstraction:

$$[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : \tau]\!] = \exists\gamma.([\![t_1 : \gamma]\!] \wedge \mathsf{def}\ x : \gamma\ \mathsf{in}\ [\![t_2 : \tau]\!])$$

Yann Régis-Gianas
yrg@irif.fr

Annotated `let`

The programmer may have provided a precise specification on `let` using a constrained type scheme. In that case, we make the constraint available in the left-hand side of `let` and require it to be satisfiable at each occurrence:

$$\frac{E \wedge E', \Gamma \vdash t_1 : \tau_1 \quad \overline{\alpha} \# \mathsf{FTV}(E, \Gamma) \quad E, \Gamma; (x : \forall \overline{\alpha}[E'].\tau_1) \vdash t_2 : \tau_2}{E, \Gamma \vdash \mathsf{let}\ x :: \forall \overline{\alpha}[E'].\tau_1 = t_1\ \mathsf{in}\ t_2 : \tau_2}$$

$$\frac{(x : \forall \overline{\alpha}[E'].\tau) \in \Gamma \qquad E \models [\overline{\alpha} \mapsto \overline{\tau}]E'}{E, \Gamma \vdash x : [\overline{\alpha} \mapsto \overline{\tau}]\tau}$$

The constraint generation rule for annotated `let` is:

$$[\![\mathsf{let}\ x :: \forall \overline{\alpha}[E'].\tau_1 = t_1\ \mathsf{in}\ t_2 : \tau_2]\!] = ([\overline{\gamma}](\forall \overline{\alpha}.E' \Rightarrow [\![t_1 : \tau_1]\!])) \wedge \mathsf{def}\ x : \forall \overline{\alpha}[E'].\tau_1\ \mathsf{in}\ [\![t_2 : \tau_2]\!]$$

where $\overline{\gamma}$ are the existentially type variables that are bound in the context.

Example

```
1  let fr =
2    fun x r ->
3      let g = fun z -> not x in
4      match r with
5      | TyBool -> g ()
6      | _ -> true
```

The standard typing constraint for this program looks like:

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{fr} : \forall \alpha_1 [\\
\quad \exists \gamma_x \gamma_r \gamma_o .\ \alpha_1 = \gamma_x \to \mathsf{repr}\ \gamma_r \to \gamma_o \wedge \\
\quad \mathsf{let}\ x : \gamma_x, r : \gamma_r\ \mathsf{in} \\
\quad \mathsf{let}\ \mathrm{g} : \forall \alpha_2 [\exists \gamma_z .\ x \preceq \mathsf{bool} \wedge \alpha_2 = \gamma_z \to \mathsf{bool}].\alpha_2\ \mathsf{in} \\
\qquad (\gamma_r = \mathsf{bool} \Rightarrow g \preceq \mathsf{unit} \to \mathsf{bool}) \wedge (\gamma_o = \mathsf{bool}) \\
].\alpha_1\ \mathsf{in} \quad \ldots
\end{array}
$$

A complete solver would defer the simplification of constraint $x \preceq \mathsf{bool}$ because the flexible type variable γ_x could be generalized globally and be refined locally.

Yann Régis-Gianas
yrg@irif.fr

Example

```
1  let fr =
2    fun x r ->
3      let g = fun z -> not x in
4      match r with
5      | TyBool -> g ()
6      | _ -> true
```

When generalization is ruled out, typing constraint looks like:

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{fr} : \forall\alpha_1[\\
\quad \exists\gamma_x\gamma_r\gamma_o.\ \alpha_1 = \gamma_x \to \mathsf{repr}\,\gamma_r \to \gamma_o \wedge \\
\quad \mathsf{let}\ x : \gamma_x, r : \gamma_r\ \mathsf{in} \\
\quad \exists\alpha_2\gamma_z.\ x \preceq \mathsf{bool} \wedge \alpha_2 = \gamma_z \to \mathsf{bool} \wedge \\
\quad \mathsf{let}\ \mathrm{g} : \alpha_2\ \mathsf{in} \\
\qquad (\gamma_r = \mathsf{bool} \Rightarrow g \preceq \mathsf{unit} \to \mathsf{bool}) \wedge (\gamma_o = \mathsf{bool}) \\
]. \alpha_1\ \mathsf{in} \quad \ldots
\end{array}
$$

In that case, γ_x must be unified with bool.

Yann Régis-Gianas
yrg@irif.fr

Example

```
1  let fr =
2    fun x r ->
3      let g = fun z -> not x in
4      match r with
5      | TyBool -> g ()
6      | _ -> true
```

After some simplifications, we have:

$$\begin{array}{l}\mathsf{let}\ \mathsf{fr} : \forall\alpha_1[\\ \quad \exists\gamma_x\gamma_r\gamma_o.\ \alpha_1 = \gamma_x \to \gamma_r \to \gamma_o \wedge\\ \quad \gamma_o = \mathsf{bool} \wedge \gamma_x = \mathsf{bool}\\ ].\alpha_1\ \mathsf{in} \quad \ldots\end{array}$$

What should we do now that we have solved the constraint of a toplevel binding?

Yann Régis-Gianas
yrg@irif.fr

Example

```
1  let fr =
2    fun x r ->
3      let g = fun z -> not x in
4      match r with
5      | TyBool -> g ()
6      | _ -> true
```

All the type variables are bounded locally. So the equality constraints must be satisfiable and the type variables can be generalized safely.

$$\mathsf{let}\ \mathsf{fr} : \forall \gamma_x[\top].\mathsf{bool} \to \mathsf{repr}\ \gamma_x \to \mathsf{bool}\ \mathsf{in}$$

...

Yann Régis-Gianas
yrg@irif.fr

A special case for toplevel `let`

Let us take a toplevel expression of the form `let x = T1 in T2`. In ML, the usual typing constraint for that expression is:

$$\text{def } x : \forall\alpha[[\![t_1 : \alpha]\!]].\alpha \text{ in } \exists\gamma.[\![t_2 : \gamma]\!]$$

All the flexible type variables that appear in constraint $[\![t_1 : \alpha]\!]$ are **local** to that constraint. As a consequence, satisfiability (and unsatisfiability) can be decided without any fear that forthcoming local assumptions could bring additional information about these variables into scope.

Thus, in a toplevel `let` binding whose typing constraint is satisfied by an mgu ϕ, all the free type variables that are not assigned by ϕ can be generalized.

A trade-off

On one hand, getting rid of let-generalization is a drastic design choice. The non homogeneous treatment of let bindings is also disturbing.

On the other hand, the authors of this approach argue that this backward incompatibility has impacted only 0.1% of the Haskell [4] libraries that are built as part of the standard GHC build process ($\sim$ 95Kloc). Besides, this design choice made possible the conception of a **modular constraint-based type inference engine** that can be instantiated to handle GADTs, type families and type classes.

[4] No benchmarks has been done on `O'Caml` libraries...

Yann Régis-Gianas
yrg@irif.fr

Type-level programming

Type families were initially aimed at implementing associated types but recently they have been used to simulate type-level computation:

```
1  type family add : * -> * -> *
2  type add zero 'a = 'a
3  type add (succ 'a) 'b = succ (add 'a 'b)
```

A lot of pressure is imposed to the simplifier because it must not only deal with the complexity of constraint solving in presence of axioms but also be efficient as a computational device!

Maybe it is time to evaluate the need for an expressive programming language at the level of types...

Yann Régis-Gianas
yrg@irif.fr

Bibliography I

Daan Leijen and Erik Meijer. "Domain specific embedded compilers". In: Proceedings of the 2nd conference on Domain-specific languages. DSL '99. Austin, Texas, United States: ACM, 1999, pp. 109–122. isbn: 1-58113-255-7. doi: http://doi.acm.org/10.1145/331960.331977. url: http://doi.acm.org/10.1145/331960.331977 (cit. on p. 7).

Barbara Liskov and Stephen Zilles. "Programming with abstract data types". In: Proceedings of the ACM SIGPLAN symposium on Very high level languages. Santa Monica, California, United States: ACM, 1974, pp. 50–59. doi: http://doi.acm.org/10.1145/800233.807045. url: http://doi.acm.org/10.1145/800233.807045 (cit. on p. 9).

Bibliography II

Hongwei Xi, Chiyan Chen, and Gang Chen. "Guarded Recursive Datatype Constructors". In: Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages. New Orleans, Jan. 2003, pp. 224–235 (cit. on p. 20).

James Cheney and Ralf Hinze. First-class phantom types. Tech. rep. Cornell University, 2003 (cit. on p. 20).

Konstantin Läufer and Martin Odersky. "An Extension of ML with First-Class Abstract Types". In: ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California. June 1992, pp. 78–91 (cit. on p. 38).

Bibliography III

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages". In: J. Funct. Program. 19.5 (2009), pp. 509–543. issn: 0956-7968. doi: http://dx.doi.org/10.1017/S0956796809007205 (cit. on p. 85).

Olivier Danvy. "Functional unparsing". In: J. Funct. Program. 8 (6 Nov. 1998), pp. 621–625. issn: 0956-7968. doi: 10.1017/S0956796898003104. url: http://dl.acm.org/citation.cfm?id=969600.969603 (cit. on p. 89).

Bibliography IV

Patricia Johann and Neil Ghani. "Foundations for structured programming with GADTs". In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. San Francisco, California, USA: ACM, 2008, pp. 297–308. isbn: 978-1-59593-689-9. doi: http://doi.acm.org/10.1145/1328438.1328475 (cit. on p. 104).

Frank Pfenning and Peter Lee. LEAP: A Language with Eval And Polymorphism. (Cit. on p. 114).

Yitzhak Mandelbaum and Aaron Stump. GADTs for the OCaml Masses. Tech. rep. (cit. on p. 125).

Bibliography V

Tom Schrijvers et al. “Complete and decidable type inference for GADTs”. In: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming. ICFP ’09. Edinburgh, Scotland: ACM, 2009, pp. 341–352. isbn: 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596599. url: http://doi.acm.org/10.1145/1596550.1596599 (cit. on pp. 134, 173, 179).

François Pottier and Yann Régis-Gianas. “Stratified type inference for generalized algebraic data types”. In: POPL. Ed. by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, 2006, pp. 232–244. isbn: 1-59593-027-2 (cit. on pp. 134, 190).

Dimitrios Vytiniotis et al. “OutsideIn(X) Modular type inference with local assumptions”. In: J. Funct. Program. 21.4-5 (2011), pp. 333–412 (cit. on pp. 134, 219, 220).

Bibliography VI

Vincent Simonet and François Pottier. “A Constraint-Based Approach to Guarded Algebraic Data Types”. In: ACM Transactions on Programming Languages and Systems (Dec. 2005). To appear (cit. on p. 163).

Yann Régis-Gianas
yrg@irif.fr