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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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

Yann Régis-Gianas
`yrg@irif.fr`

2019-01-11

Plan

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

Type inference in ML extended with GADTs

Why is type inference difficult?

Stratified type inference

`OutSideIn(X)`

Plan

Type inference in ML extended with GADTs

Why is type inference difficult?

Stratified type inference

`OutSideIn(X)`

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

Dealing with a difficult type inference problem

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

▶ 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 [**Schrijvers09**].
2. The Stratified Type Inference approach [**PottierRG06**].
3. The modular OutsideIn(X) without let-generalization approach [**Vytiniotis11**]

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
```

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

The loss of principality

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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:

▶ $\forall \alpha \beta. \, \text{eq} \, \alpha \, \beta \to \alpha \to \beta$

▶ $\forall \alpha \beta. \, \text{eq} \, \alpha \, \beta \to \beta \to \alpha$

▶ $\forall \alpha \beta \gamma. \, \text{eq} \, \alpha \, \beta \to \gamma \to \gamma$

Problem 1: What are the available type equalities?

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

```
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.

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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?

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

```
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?

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

```
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.

Problem 3: Is that dead code or not?

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

```
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.

Four inter-connected problems

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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):

$$C ::= \dots \mid C \Rightarrow C$$

New constraint generation rules

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Recall the rules for ML:

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

What would be the constraint generation rule for pattern matching?

New constraint generation rules

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

$$\llbracket \text{match } T \text{ with } p_1 \Rightarrow T_1 \mid \ldots \mid p_n \Rightarrow T_n : \tau \rrbracket =$$

New constraint generation rules

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs

Why is type inference difficult?

Stratified type inference

`OutSideIn(X)`

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

New constraint generation rules

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

$$\llbracket \text{match } T \text{ with } p_1 \Rightarrow T_1 \mid \ldots \mid p_n \Rightarrow T_n : \tau \rrbracket =$$
$$\exists \gamma. \; \llbracket T : \gamma \rrbracket \qquad \text{The scrutinee has type } \gamma.$$
$$\wedge \; (\llbracket p_1 : \gamma \rrbracket \qquad \text{The pattern has type } \gamma.$$

New constraint generation rules

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

$$\llbracket \text{match } T \text{ with } p_1 \Rightarrow T_1 \mid \ldots \mid p_n \Rightarrow T_n : \tau \rrbracket \qquad =$$
$$\exists \gamma. \ \llbracket T : \gamma \rrbracket \qquad \qquad \text{The scrutinee has type } \gamma.$$
$$\land (\llbracket p_1 : \gamma \rrbracket \qquad \qquad \text{The pattern has type } \gamma.$$
$$\land \ \forall \overline{\beta}_1. \ \text{def } \Gamma_1 \ \text{in } C_1 \Rightarrow \qquad \text{The local assumptions...}$$

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

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

New constraint generation rules

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

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

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

New constraint generation rules

$\llbracket \text{match } T \text{ with } p_1 \Rightarrow T_1 \mid \ldots \mid p_n \Rightarrow T_n : \tau \rrbracket$ =
$\quad \exists \gamma. \; \llbracket T : \gamma \rrbracket$ The scrutinee has type γ.
$\quad\quad \wedge \; (\llbracket p_1 : \gamma \rrbracket$ The pattern has type γ.
$\quad\quad\quad \wedge \; \forall \overline{\beta}_1. \; \text{def } \Gamma_1 \text{ in } C_1 \Rightarrow$ The local assumptions...
$\quad\quad\quad\quad \llbracket T_1 : \tau \rrbracket)$ ...under which the body is typed.

$\cdots$

$\quad\quad \wedge \; (\llbracket p_n : \gamma \rrbracket \; \wedge \; \forall \overline{\beta}_n. \; \text{def } \Gamma_n \text{ in } C_n \Rightarrow \llbracket T_n : \tau \rrbracket)$

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

How to extract the local assumptions?

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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

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

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

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.

How to extract the local assumptions?

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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

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

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

What is the typing constraint of a pattern?

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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

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

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

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

What is the typing constraint of a pattern?

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

In our formulation, we have:

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

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

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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):

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

How to solve such an implication constraint?

Solving an implication constraint by hand

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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

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.

Solving an implication constraint by hand

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

$$\text{let } f : \forall \gamma.[$$
$$\exists \gamma_{xr}.$$
$$\gamma = \gamma_{xr} \to \text{repr } \gamma_{xr} \to \text{string}$$
$$\land \gamma_{xr} = \text{int} \Rightarrow \gamma_{xr} = \text{int}$$
$$\land \forall \beta \alpha. \gamma_{xr} = \alpha + \beta \Rightarrow \exists \gamma_1 \gamma_2. \gamma_{xr} = \gamma_1 + \gamma_2$$
$$\land \forall \beta \alpha. \gamma_{xr} = \alpha \times \beta \Rightarrow \exists \gamma_1 \gamma_2. \gamma_{xr} = \gamma_1 \times \gamma_2$$
$$].\gamma$$
$$\text{in true}$$

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.)

Solving an implication constraint by hand

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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

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

$$\text{let } f : \forall \gamma_{xr}. \; \gamma_{xr} \rightarrow \text{repr } \gamma_{xr} \rightarrow \text{string in true}$$

Another path

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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

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

Another path

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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

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.

Another path

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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

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

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

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.

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Another path

$$\text{let } f : \forall \gamma.[$$
$$\exists \gamma_x \gamma_r.$$
$$\gamma = \gamma_x \rightarrow \text{repr } \gamma_r \rightarrow \text{string}$$
$$\land \gamma_x = \text{int} \land \gamma_r = \text{string}$$
$$\land \forall \beta \alpha.\text{false} \Rightarrow \exists \gamma_1 \gamma_2.\gamma_x = \gamma_1 + \gamma_2$$
$$\land \forall \beta \alpha.\text{false} \Rightarrow \exists \gamma_1 \gamma_2.\gamma_x = \gamma_1 \times \gamma_2].\gamma$$
$$\text{in true}$$

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

$$\text{let } f : \text{int} \rightarrow \text{repr string} \rightarrow \text{string in true}$$

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`.

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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 [**simonet-pottier-hmg-toplas**].

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

Theorem (Soundness [**simonet-pottier-hmg-toplas**])
$[\![e : \tau]\!], \Gamma \vdash e : \tau$ holds.

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

Example

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

```
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:

$$\forall \alpha \beta [$$
$$\alpha = \mathsf{int} \Rightarrow \beta = \mathsf{int}$$
$$\wedge \forall \beta_1 \beta_2.\ \alpha = \beta_1 + \beta_2 \Rightarrow \exists \gamma_1 \gamma_2.\ \beta = \gamma_1 + \gamma_2$$
$$\wedge \forall \beta_1 \beta_2.\ \alpha = \beta_1 \times \beta_2 \Rightarrow \exists \gamma_1 \gamma_2.\ \beta = \gamma_1 \times \gamma_2$$
$$].\ \beta \rightarrow \mathsf{repr}\ \alpha \rightarrow \mathsf{string}$$

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.

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

Untractable type inference

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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?

A constraint generation rule for user-annotated functions

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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

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

To adequately express the user type annotation, we underlined the type variable α to stress the requirement that this type variable cannot be unified globally with another type. Indeed, α is a

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

A class of easy-to-solve constraints

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

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

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 $C \equiv$ false, we are done.

▶ If there is a most general unifier for C, apply it to D and try to solve D.

When does this approach break?

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Let us transform this program using an η—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.

Let us try harder

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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

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

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

Eventually, the rewritten constraint only contains rigid implications. 28/84

How long can the typing constraints be suspended?

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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.

Another example

Consider now the following example [**Schrijvers09**]:

```
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`.)

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

The typing constraint of f1

$$\text{let } f1 : \forall \alpha [$$
$$\exists \gamma_1 \gamma_2 . \alpha = \gamma_1 \to \gamma_2$$
$$\land \exists \gamma_3 . \gamma_1 = t \gamma_3$$
$$\land \forall \gamma_4 . (\gamma_1 = t \gamma_4 \land \gamma_4 = \mathsf{bool} \Rightarrow \gamma_2 = \mathsf{bool})$$
$$].\alpha \text{ in } \dots$$

First, we simplify non implication constraints.

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

31/84

The typing constraint of $f1$

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

$$\text{let } f1 : \forall \gamma_2 \gamma_3 [$$
$$\gamma_3 = \text{bool} \Rightarrow \gamma_2 = \text{bool})$$
$$].t \ \gamma_3 \rightarrow \gamma_2 \text{ in } \dots$$

The following two incompatible assignments satisfy the implication:

▶ $\gamma_2 \mapsto \text{bool}$, yielding $f1 : \forall \alpha.t \ \alpha \rightarrow \text{bool}$

▶ $\gamma_2 \mapsto \gamma_3$, yielding $f1 : \forall \alpha.t \ \alpha \rightarrow \alpha$

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`

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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

Again, we first simplify non implication constraints.

Typing constraint of f2

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

$$\begin{aligned}
\text{let } f2 : \forall \gamma_2 \gamma_3 [& \\
& \gamma_3 = \text{bool} \Rightarrow \gamma_2 = \text{bool}) \\
& \wedge \gamma_2 = \text{bool}) \\
]. \ t \ \gamma_3 \to \gamma_2 \text{ in } \dots &
\end{aligned}$$

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.

Typing constraint of `f2`

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

$$\text{let } f2 : \forall \gamma_2 \gamma_3 [$$
$$\gamma_3 = \text{bool} \Rightarrow \text{bool} = \text{bool})$$
$$]. \, t \, \gamma_3 \rightarrow \text{bool in } \ldots$$

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

$$f2 : \forall \alpha.t \, \alpha \rightarrow \text{bool}$$

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 [**Schrijvers09**] 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:

$$\text{let } f1 : \forall\alpha[$$
$$\exists\gamma_1\gamma_2.\alpha = \gamma_1 \rightarrow \gamma_2$$
$$\wedge \exists\gamma_3. \ \gamma_1 = t \ \gamma_3$$
$$\wedge [\gamma_2 \ \gamma_1](\forall\gamma_4. \ (\gamma_1 = t \ \gamma_4 \ \wedge \ \gamma_4 = \text{bool}) \Rightarrow \gamma_2 = \text{bool})$$
$$].\alpha \text{ in } \dots$$

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

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

33/84

Simplifying implication constraints

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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.

Example

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

$$\text{let } f1 : \forall \gamma_2 \gamma_3 [$$
$$[\gamma_3 \ \gamma_2](\gamma_3 = \text{bool} \Rightarrow \gamma_2 = \text{bool})$$
$$].t \ \gamma_3 \rightarrow \gamma_2 \text{ in } \dots$$

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

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Example

In the second case, the typing constraint:

$$\text{let } f2 : \forall \gamma_2 \gamma_3 [$$
$$[\gamma_3 \; \gamma_2]\gamma_3 = \text{bool} \Rightarrow \gamma_2 = \text{bool})$$
$$\wedge \gamma_2 = \text{bool})$$
$$]. \, \flat \, \gamma_3 \rightarrow \gamma_2 \text{ in } \ldots$$

is simplified into

$$\text{let } f2 : \forall \gamma_2 \gamma_3 [$$
$$[\gamma_3 \; \gamma_2]\gamma_3 = \text{bool} \Rightarrow \text{bool} = \text{bool})$$
$$]. \, \flat \, \gamma_3 \rightarrow \text{bool in } \ldots$$

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

Metatheorems

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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.

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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:

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

The typing constraint looks like:

$$
\begin{aligned}
&\text{let } f : \forall \gamma_1 \gamma_2 [\\
&\quad \text{let } h : \forall \gamma_3 [[\gamma_1 \ \gamma_3](\gamma_1 = \text{int} \Rightarrow \gamma_3 = \text{int})].\gamma_3 \text{ in} \\
&\quad \gamma_2 = \text{string} \\
&]. \ \gamma_1 \rightarrow \gamma_2
\end{aligned}
$$

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

A declarative definition of completeness for `OutsideIn`

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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`

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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?

A declarative definition of completeness for `OutsideIn`

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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.

Explaining accepted programs to the lambda programmer

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

"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."

Summary

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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.

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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 [**PottierRG06**] 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

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML extended with GADTs

Why is type inference difficult?

Stratified type inference

`OutSideIn(X)`

MLGX

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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.

Restriction 1: Scrutinee must be annotated

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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:

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

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

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) -> (* ∃β₁β₂.α = β₁ + β₂ *) ...
6      | TyProd (ty1, ty2) -> (* ∃β₁β₂.α = β₁ × β₂ *) ...
```

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

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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:

$$\theta \ ::= \ \exists \overline{\gamma}.\tau$$
$$\zeta \ ::= \ \exists \overline{\gamma}.\sigma$$

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 annotations, that is why we extend the syntax of patterns to:

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

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Type checking rules for pattern matching

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

X-Match
$$E, \Gamma \vdash (t : \theta) : \tau_1$$
$$\forall i, E, \Gamma \vdash (p_i : \theta) \Rightarrow t_i : \tau_1 \rightarrow \tau_2$$
$$\overline{E, \Gamma \vdash \mathsf{match}\ (t : \theta)\ \mathsf{with}\ p_1 \Rightarrow t_1 \mid \ldots \mid p_n \Rightarrow t_n : \tau_2}$$

X-Branch
$$p : \varepsilon\ \overline{\tau}_1\ \overline{\tau}_2' \vdash (\overline{\beta}, E', \Gamma')$$
$$E \wedge E', \Gamma\Gamma' \vdash t : \tau$$
$$\overline{E, \Gamma \vdash (p : \exists \overline{\gamma}.\varepsilon\ \_\ \overline{\tau}_2') \Rightarrow t : \varepsilon\ \overline{\tau}_1\ \_ \rightarrow \tau}$$

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

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

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

```
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
```

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.

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

MLGI is an implicit version of MLGX

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

What is the link between MLGI and MLGX?

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

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

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.

Explaining MLGX to the lambda programmer

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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.

Type inference for MLGX

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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 $\forall \overline{\gamma}.\tau_1 \rightarrow \tau_2$, that computationally behaves like the identity.

More formally...

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

$$\llbracket E \vdash (t : \exists \overline{\gamma}.\ \tau_1 \rhd \tau_2) : \tau \rrbracket$$
$$= \exists \overline{\gamma}.(\llbracket E \vdash t : \tau_1 \rrbracket) \wedge \tau_2 = \tau)$$

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

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

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

Complete type inference for MLGX

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

Yann Régis-Gianas
`yrg@irif.fr`

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
`OutSideIn(X)`

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.

Example

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Here is the function `f` in MLGX:

```
1  let f = forall 'a.
2    fun x r ->
3      match (r : 'a repr) with
4      | TyInt -> string_of_int (x : 'a :> int)
5      | TySum ['b1 'b2] (ty1, ty2) ->
6          (match (x : 'a :> 'b1 + 'b2) with
7           | Left x1 -> "L"
8           | Right x2 -> "R")
9      | 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.

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

Yann Régis-Gianas
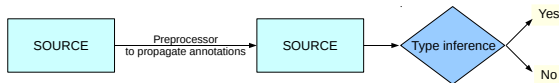yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Heterogeneous difficulties

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

```
1  let f : forall 'a. 'a -> 'a repr -> string =
2    fun x  r ->
3      match (r : 'a repr) with
4      | TyInt -> string_of_int (x : 'a :> int)
5      | TySum 'b1 'b2 (ty1, ty2) ->
6          (match (x : 'a :> 'b1 + 'b2) with
7           | Left x1 -> "L"
8           | Right x2 -> "R")
9      | 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.

Stratified type inference

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

Yann Régis-Gianas
yrg@irif.fr

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.

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

On the incompleteness of the front-end

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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."

Picturing local type inference

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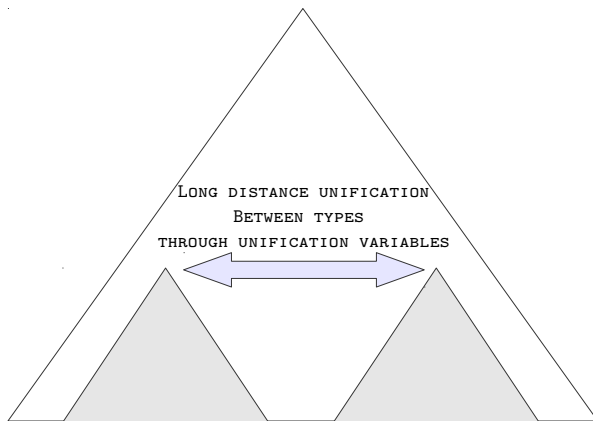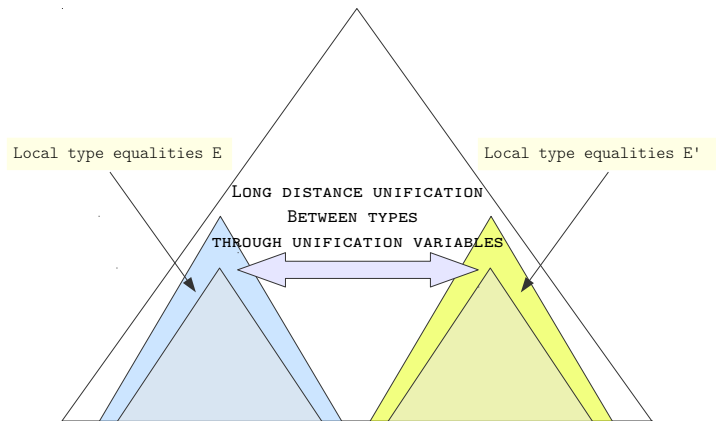Tractable typing
– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

ML type inference

Picturing local type inference

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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

Picturing local type inference

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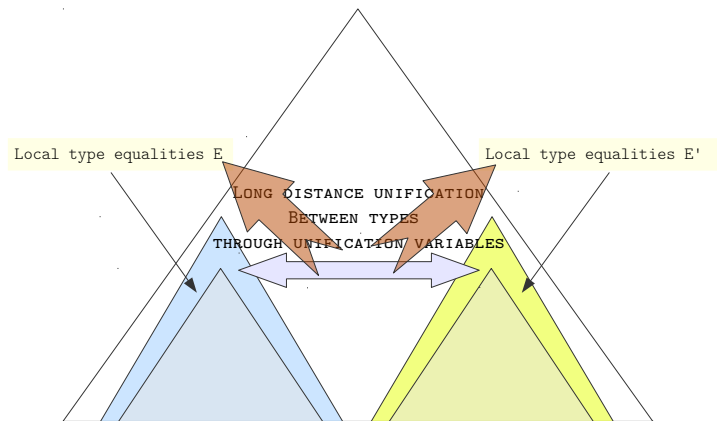– Dependently-typed
programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Local type equalities E   Local type equalities E'

LONG DISTANCE UNIFICATION
BETWEEN TYPES
THROUGH UNIFICATION VARIABLES

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

Picturing local type inference

Rich types,
Tractable typing
– Dependently-typed
programming
languages –
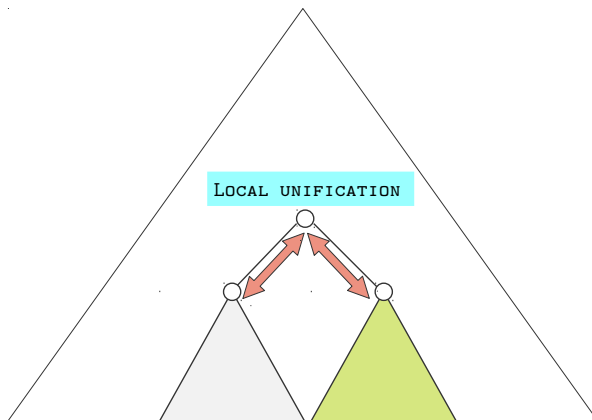
Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Local type inference uses local unification.

Picturing local type inference

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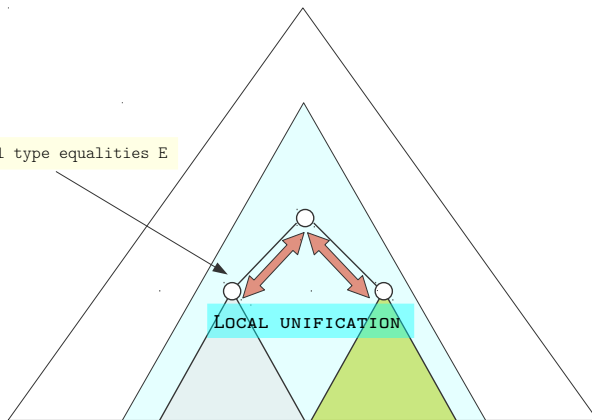programming
languages –

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Local type equalities E

LOCAL UNIFICATION

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

Example

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

```
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.

Precautious propagations

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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 $E = \text{true}$, then it cannot be unified into α.

How to formalize that idea?

Plan

Type inference in ML extended with GADTs

Why is type inference difficult?

Stratified type inference

OutSideIn(X)

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Recreational break

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Recreational break

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference

OutSideIn(X)

Another design choice: Dropping let generalization

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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.

Example (inspired by [**Vytiniotis11**])

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs

Why is type inference difficult?

Stratified type inference

OutSideIn(X)

```
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
```

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?

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

Example (inspired by [**Vytiniotis11**])

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

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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
```

No! This program should be rejected.

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

```
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.)

Let us generalize the idea of non generalization

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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

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

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

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

Annotated **let**

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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:

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

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

The constraint generation rule for annotated **let** is:

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

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
```

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

The standard typing constraint for this program looks like:

$$\text{let fr} : \forall\alpha_1[$$
$$\quad \exists\gamma_x\gamma_r\gamma_o. \; \alpha_1 = \gamma_x \to \text{repr } \gamma_r \to \gamma_o \wedge$$
$$\quad \text{let } x : \gamma_x, r : \gamma_r \text{ in}$$
$$\quad \text{let g} : \forall\alpha_2[\exists\gamma_z. \; x \preceq \text{bool} \wedge \alpha_2 = \gamma_z \to \text{bool}].\alpha_2 \text{ in}$$
$$\quad\quad (\gamma_r = \text{bool} \Rightarrow g \preceq \text{unit} \to \text{bool}) \wedge (\gamma_o = \text{bool})$$
$$].\alpha_1 \text{ in} \quad \dots$$

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

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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:

$$
\begin{aligned}
&\text{let fr} : \forall \alpha_1 [\\
&\quad \exists \gamma_x \gamma_r \gamma_o.\ \alpha_1 = \gamma_x \rightarrow \text{repr } \gamma_r \rightarrow \gamma_o \wedge \\
&\quad \text{let } x : \gamma_x, r : \gamma_r \text{ in} \\
&\quad \exists \alpha_2 \gamma_z.\ x \preceq \text{bool} \wedge \alpha_2 = \gamma_z \rightarrow \text{bool} \wedge \\
&\quad \text{let } g : \alpha_2 \text{ in} \\
&\quad \quad (\gamma_r = \text{bool} \Rightarrow g \preceq \text{unit} \rightarrow \text{bool}) \wedge (\gamma_o = \text{bool}) \\
&].\alpha_1 \text{ in } \quad \dots
\end{aligned}
$$

In that case, γ_x must be unified with bool.

Example

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

```
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:

$$\text{let } fr : \forall \alpha_1 [$$
$$\exists \gamma_x \gamma_r \gamma_o.\ \alpha_1 = \gamma_x \to \gamma_r \to \gamma_o \land$$
$$\gamma_o = \text{bool} \land \gamma_x = \text{bool}$$
$$].\alpha_1 \text{ in } \quad \dots$$

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

Example

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
  Why is type inference difficult?
  Stratified type inference
  OutSideIn(X)

```
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.

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

...

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.

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

A trade-off

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs
Why is type inference difficult?
Stratified type inference
OutSideIn(X)

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 [1] 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.

---

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

Bibliography I

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

Yann Régis-Gianas
yrg@irif.fr

Type inference in ML
extended with GADTs

Why is type inference difficult?

Stratified type inference

OutSideIn(X)