# Towards proved programs
# Part II: Type inference in ML with GADTs

MPRI 2-4-2 (Version Fri Jan 30 12:20:08 CET 2015 )

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

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

# Contents

# Contents

# 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 [Schrijvers et al., 2009].
  2. The Stratified Type Inference approach [Pottier and Régis-Gianas, 2006].
  3. The modular OutsideIn(X) without let-generalization approach [Vytiniotis et al., 2011]

# The loss of principality

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

```
let cast : ? =
  fun p x →
    match p with
    | Refl → x
```

# The loss of principality

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

```
let cast : ? =
  fun p x →
    match p with
    | Refl → x
```

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

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

# Problem 1: What are the available type equalities?

```
let cast : ? =
  fun p x →
    match p with
    | 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?

```
let cast : ? =
  fun p x →
    match p with
    | Refl → x
```

Assume that, at some point, the type `eq` $\alpha$ $\beta$ of `p` is known to the type inference engine. This implies that the type equality $\alpha = \beta$ 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?

```
let cast : ? =
  fun p x →
    match p with
    | Refl → x
```

On the one hand, *if we assume that this type equality is used*, it is equivalent, **locally**, to assign the type $\alpha$ or the type $\beta$ 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?

```
let cast : ? =
  fun p x →
    match p with
    | 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 $\alpha$ and $\beta$.

# Problem 3: Is that dead code or not?

```
let succ : ? =
  fun r x →
    1 + match r with
    | TyInt → x
    | 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

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):

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

# New constraint generation rules

Recall the rules for ML:

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

What would be the constraint generation rule for pattern matching?

# New constraint generation rules

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

# 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

$$
\begin{aligned}
&[\![\text{match } T \text{ with } p_1 \Rightarrow T_1 \mid \ldots \mid p_n \Rightarrow T_n : \tau]\!] = \\
&\quad \exists \gamma.\ [\![T : \gamma]\!] && \text{The scrutinee has type } \gamma. \\
&\qquad \wedge\ ([\![p_1 : \gamma]\!] && \text{The pattern has type } \gamma.
\end{aligned}
$$

# New constraint generation rules

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

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

# New constraint generation rules

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

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

# New constraint generation rules

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

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

# How to extract the local assumptions?

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

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

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

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

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

# 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.

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

# Example

```
let f =
  fun x r →
    match r with
    | TyInt → string_of_int x
    | TySum (ty1, ty2) → (match x with Left x1 → "L" | Right x2 → "R")
    | 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?

# Solving an implication constraint by hand

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

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

# Solving an implication constraint by hand

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

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

1. Compute $\phi$, 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

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

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

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

# Another path

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

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

# Another path

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

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

# Another path

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

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.

# Another path

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

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

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

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`.

# 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 and Pottier, 2005].

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

**Theorem (Soundness [Simonet and Pottier, 2005])**

$[\![e : \tau]\!], \Gamma \vdash e : \tau$ *holds.*

**Theorem (Completeness [Simonet and Pottier, 2005])**

*If* $C, \Gamma \vdash e : \forall \overline{\alpha}[D].\tau$ *then* $C \models \forall \overline{\alpha}.\ D \Rightarrow$ *def* $\Gamma$ *in* $[\![e : \tau]\!]$

# Example

```
let f =
  fun x r →
    match r with
    | TyInt → string_of_int x
    | TySum (ty1, ty2) → (match x with Left x1 → "L" | Right x2 → "R")
    | TyProd (ty1, ty2) → "P"
```

admits a most general constrained type scheme which is:

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

# 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.

# 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`:

```
let f : ∀α. α → α repr → string =
  fun x r →
    match r with
    | TyInt → string_of_int x
    | TySum (ty1, ty2) → (match x with Left x1 → "L" | Right x2 → "R")
    | 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

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 $\alpha$ to stress the requirement that this type variable cannot be unified globally with another type. Indeed, $\alpha$ is a polymorphic type parameter of the function. (Recall: Universally quantified type variables are said to be **rigid**.)

# A class of easy-to-solve constraints

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

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

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 $C \equiv$ 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?

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

```
let f : ∀α. α → α repr → string
  fun y → (fun x r →
    match r with
    | TyInt → string_of_int x
    | TySum (ty1, ty2) → (match x with Left x1 → "L" | Right x2 → "R")
    | 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

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

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

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

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.

# Another example

Consider now the following example [Schrijvers et al., 2009]:

```
type t α =
  | K1 : int → bool t
  | K2 : ∀ α. α list → α t

let f1 = function
  | K1 n → n > 0

let f2 = function
  | K1 n → n > 0
  | K2 xs → is_nil xs
```

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

# The typing constraint of `f1`

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

First, we simplify non implication constraints.

# The typing constraint of `f1`

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

The following two incompatible assignments satisfy the implication:

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

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`

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

Again, we first simplify non implication constraints.

# Typing constraint of `f2`

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

At some point, we have simplified the second implication, leading to an unconditional assignment of $\gamma_2$. This time the information about $\gamma_2$ 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`

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

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

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

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 [Schrijvers et al., 2009] 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:

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

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

# 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 $\phi$, whose domain may contain some untouchables.
2. Apply $\phi$ to $C_2$ and simplify $C_2$.
3. Check that the resulting mgu $\phi'$ **does not assign** one of the untouchables.

# Example

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

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 $\gamma_2$, this constraint is rejected.

# Example

In the second case, the typing constraint:

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

is simplified into

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

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

# 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 ([Xi et al., 2003] was not):

**Theorem (Conservative extension of ML)**

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

# 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 α t = K : int t
(* Function f admits the following principal type scheme:  *)
let f : ∀ α. α t → string =
  fun x → let h = match x with K → 42 in
    "foobar"
```

The typing constraint looks like:

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

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`

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?

# 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.

# 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."*

# 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 [Pottier and Régis-Gianas, 2006] 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.

# Contents

# 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.

# 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:

```
let f : ∀α. α → α 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:

```
let f : ∀α. α → α repr → string =
  fun x r →
    match (r : α repr) with
    | TyInt → (* α = int *) ...
    | TySum (ty1, ty2) → (* ∃β1β2.α = β1 + β2*) ...
    | TyProd (ty1, ty2) → (* ∃β1β2.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 $\theta$ (resp. a type scheme annotation $\zeta$), the type variables $\overline{\gamma}$ bind inside $\tau$ (resp. $\sigma$). 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:

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

We are now ready to write type-checking rules for pattern matching in MLGX.

# 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

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

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

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

```
type (α, 'n) list =
| Nil :  (α, zero) list
| Cons :  ∀ α n. α × (α, 'n) list → (α, 'n succ) list

let head = forall n.
   match (l :  exists α. (α, n succ) list) with Cons (x, _) → x
```

# Restriction 2 : Coercions must be explicited

In MLGX, **the usage of type equalities are explicited** using **coercions** $\kappa$ :

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

where $\overline{\gamma}$ binds into $\tau_1$ and $\tau_2$.

The rule for coercion replaces the implicit conversion by allowing a term of type $\tau_1$ to be seen as a term of type $\tau_2$ 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 $\kappa$ can be checked orthogonally to t.

# 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:

$$\frac{t \equiv t' \qquad \overline{\alpha} \# \mathbf{ftv}(t)}{t \equiv \forall \overline{\alpha}.t'} \qquad \frac{t \equiv t'}{t \equiv (t' : \theta)} \qquad \frac{t \equiv t'}{t \equiv (t' : \kappa)}$$

**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.

# 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.

# 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 $\text{cast}_{\tau_1 \triangleright \tau_2}$ of type $\forall\overline{\gamma}.\tau_1 \to \tau_2$, that computationally behaves like the identity.

# More formally...

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

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

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

# Complete type inference for MLGX

**Theorem (MLGX has principal types)**

*$\phi$ 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

Here is the function `f` in MLGX:

```
let f = forall α.
  fun x r →
    match (r : α repr) with
    | TyInt → string_of_int (x : α ▷ int)
    | TySum β1 β2 (ty1, ty2) →
        (match (x : α ▷ β1 + β2) with Left x1 → "L" | Right x2 → "R")
    | TyProd β1 β2 (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.

# Heterogeneous difficulties

We would like to write the more concise annotation in red and deduce automatically the annotation in green:

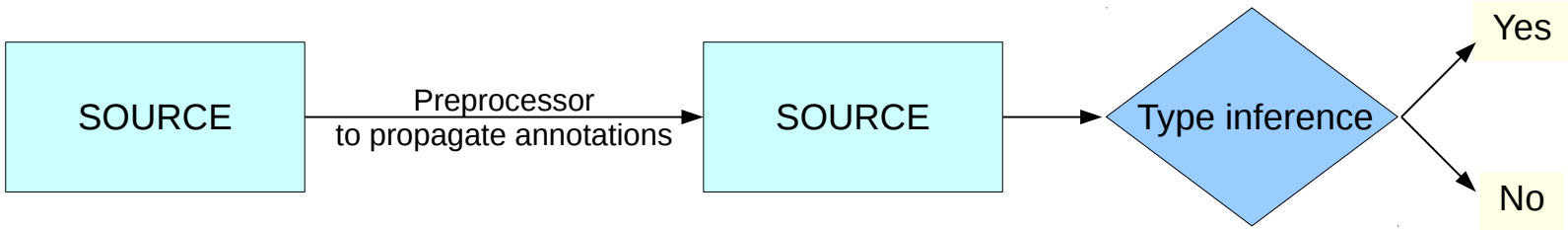
```
let f : ∀α.α → repr α → string =
  fun x r →
    match (r : repr α) with
    | TyInt → string_of_int (x : α ▷ int)
    | TySum β1 β2 (ty1, ty2) →
        (match (x : α ▷ β1 + β2) with Left x1 → "L" | Right x2 → "R")
    | TyProd β1 β2 (ty1, ty2) → "P"
```

. . . 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

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.



# On the incompleteness of the front-end

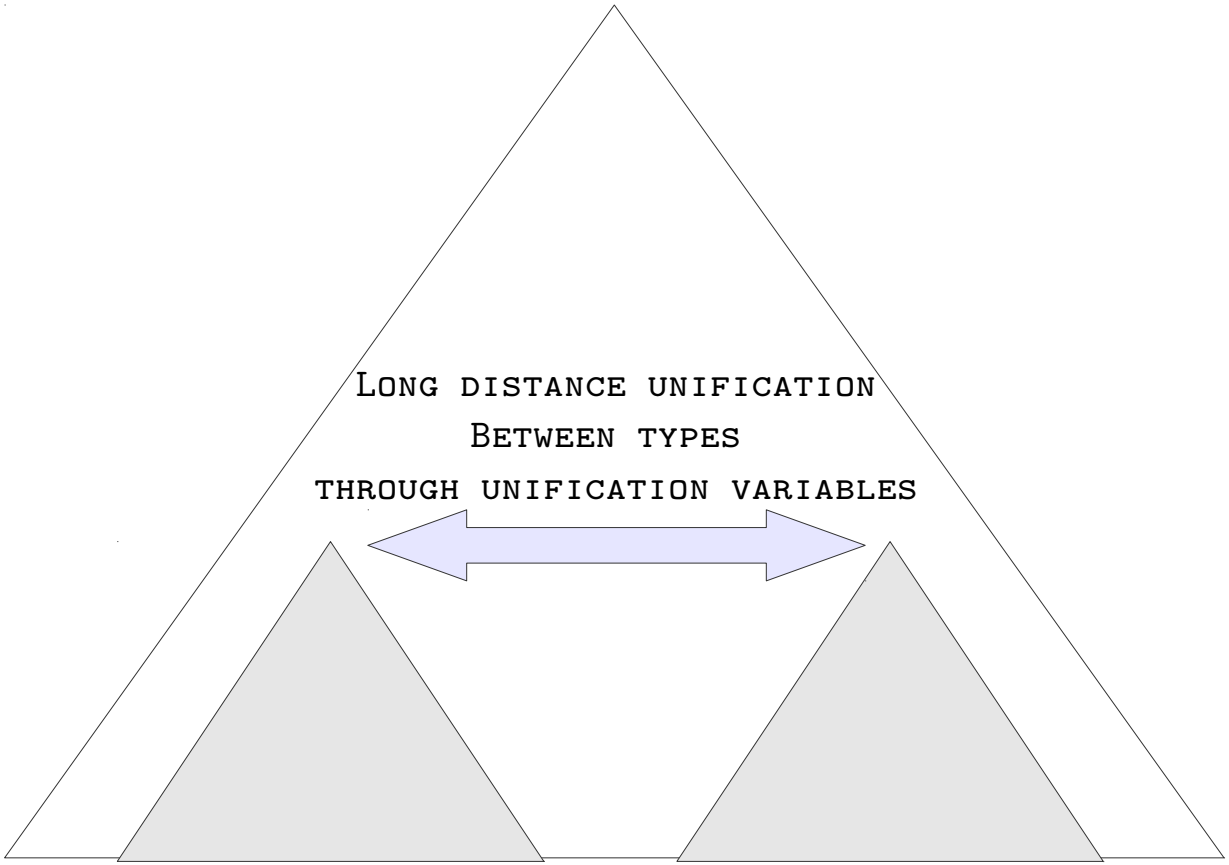
This algorithm is clearly incomplete with respect to MLGI: starting with a term without any annotations, it will not be able to reconstruct the programmer's choices by magic. Yet, we want this propagation to be **sound** and as simple and as **predictable** as possible.

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** [Pierce and Turner, 1998]:

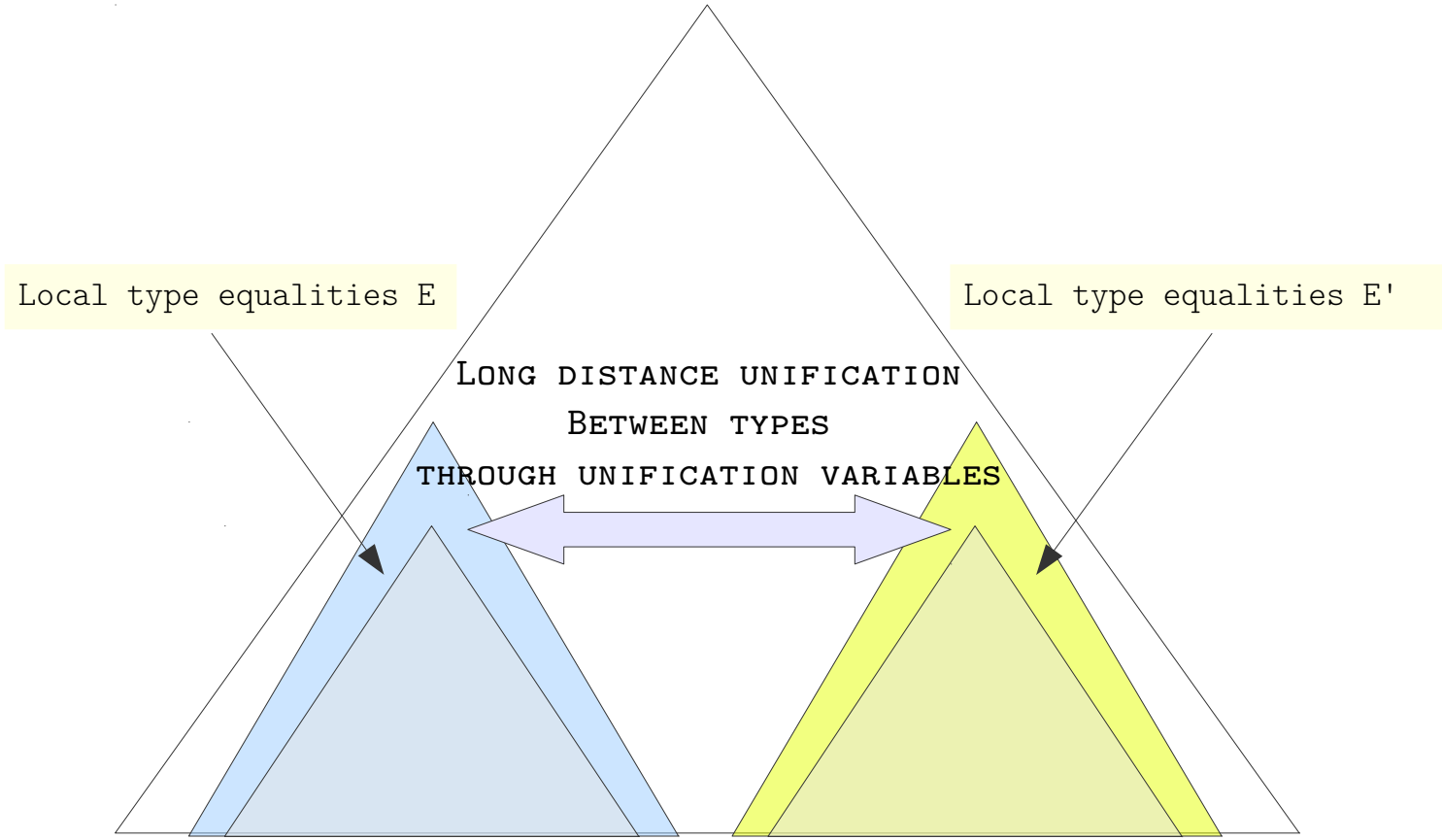
> *"Missing type annotations [should be] recovered using only information from adjacent nodes in the syntax tree, without long-distance constraints such as unification variables."*

# Picturing local type inference



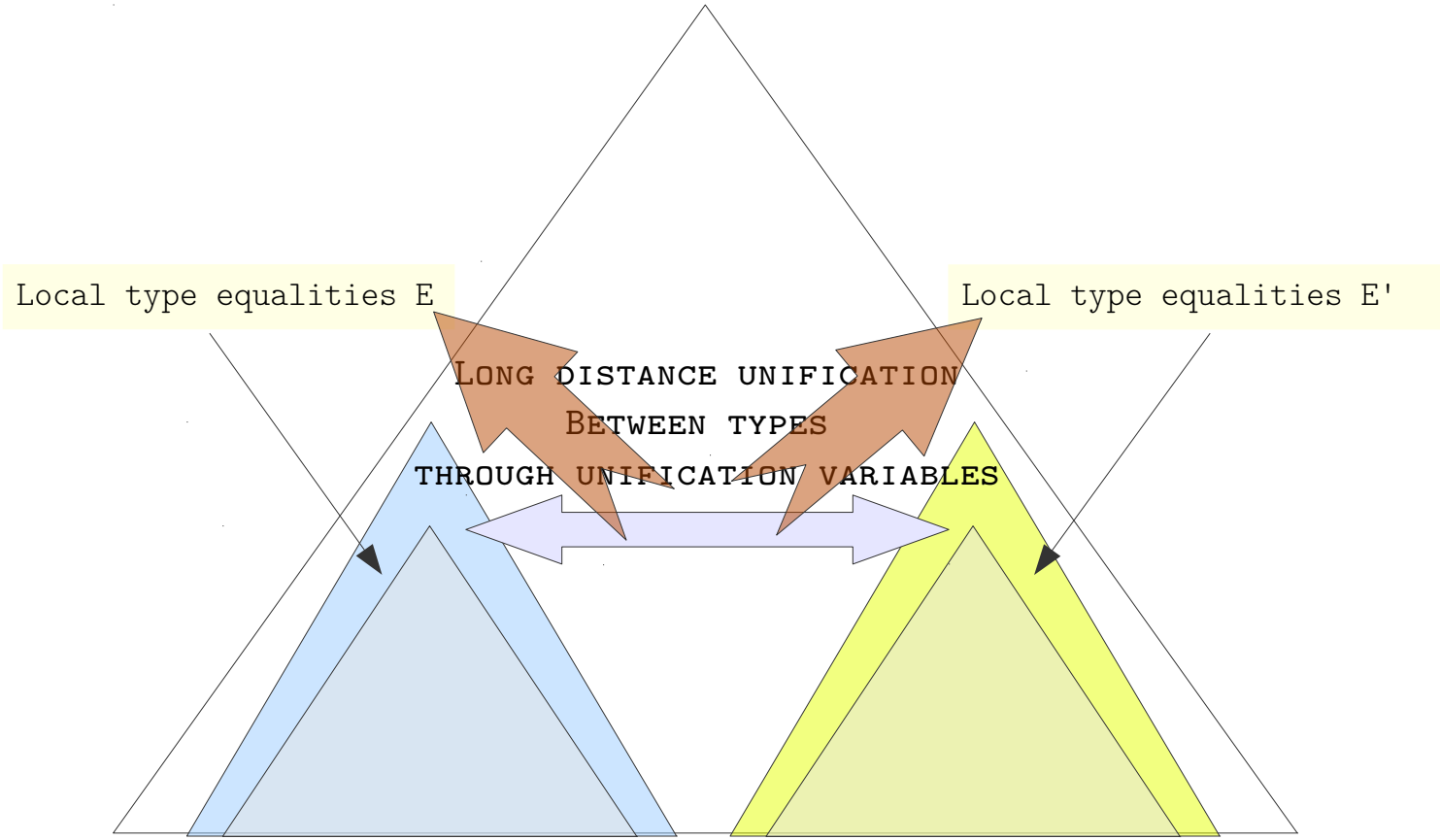

ML type inference

# Picturing local type inference



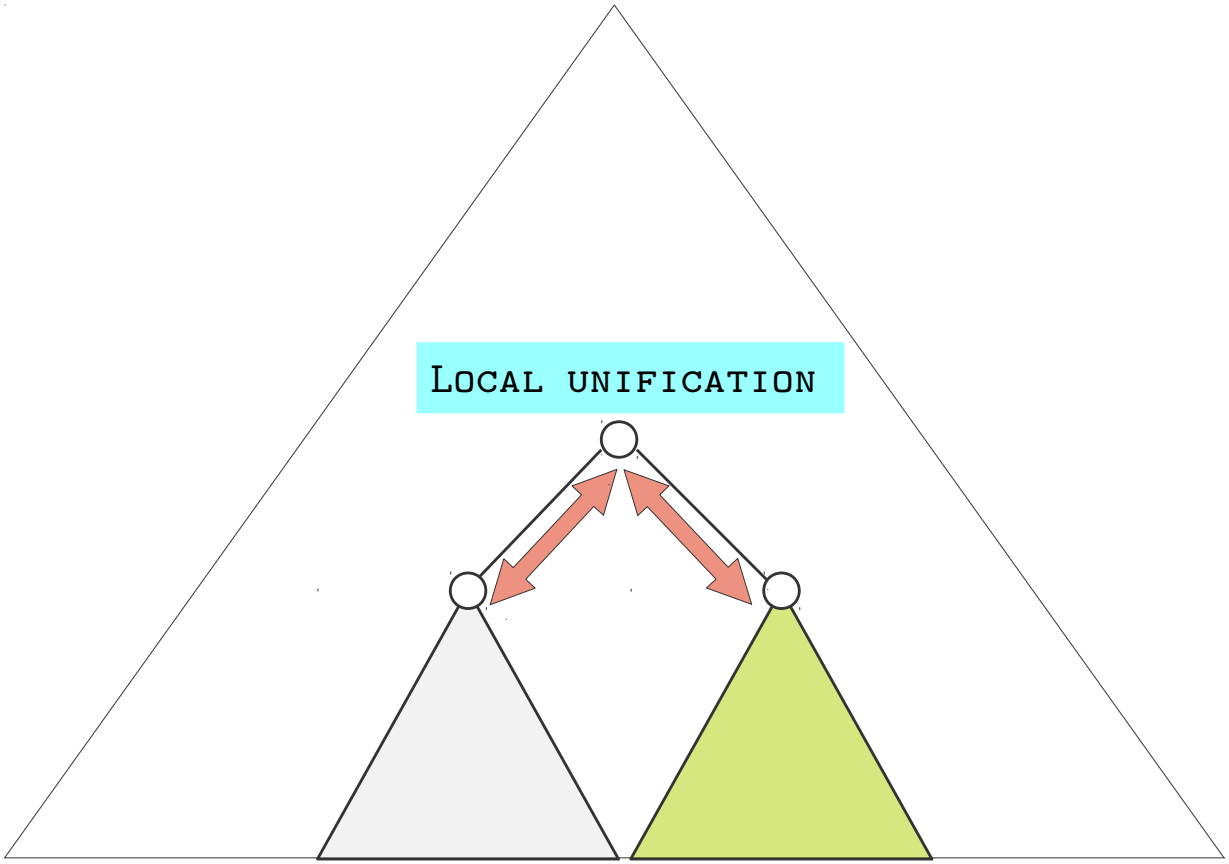

ML type inference with GADTs needs unification modulo E and E'...

# Picturing local type inference



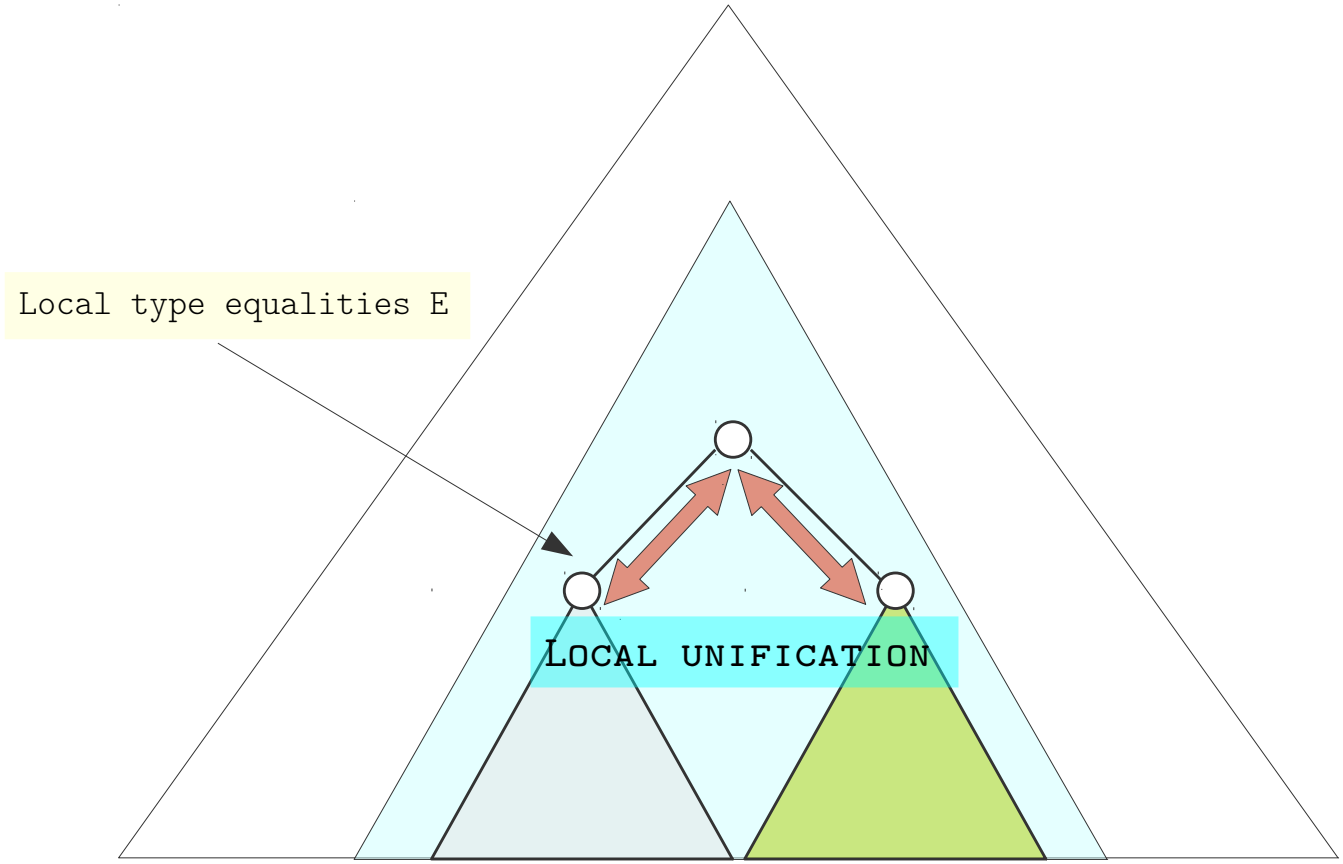

GADTs need unification modulo E and E'..., which depend on unification!

# Picturing local type inference




Local type inference uses local unification.

# Picturing local type inference




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

# Example

```
type α t =
| T1 : int → int t
| T2 : α list → α t

let apply = forall α.
  fun (x : α) p (t : α repr) →
    match t with
    | T1 n → p x ∧ n > 0
    | T2 l → (p : α → bool) (List.hd xs)
```

The user annotation says that `x : α`. Thus, in the first branch, in expression "`p x`", the variable `p` is applied to an argument that have both types $\alpha$ 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

Local assumptions define **equivalence classes** between types. At one particular moment, the algorithm that propagates the annotation may hold **a particular representant** $\tau$ 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 $\tau$ 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 $\alpha$.) Yet, if $\gamma \times \beta_2$ was written under $E = \text{true}$, then it cannot be unified into $\alpha$.

How to formalize that idea?

# Local type inference based on shapes

The idea is to introduce **shapes** which are **approximations** of types.

$$s ::= \overline{\gamma}.\tau$$

The type variables $\overline{\gamma}$ are *flexible* variables. A flexible type variable represents a type that is either unknown (from the type propagation perspective) or is a polymorphic type variable.

The free type variables of shapes are interpreted as rigid: they cannot be unified with other types. In other words, the only allowed unification between shapes are local, as shown by the instantiation relation between shapes:

$$\frac{\overline{\gamma}_2 \# \mathbf{ftv}\,(\overline{\gamma}_1, \tau_2)}{\overline{\gamma}_1.\tau_1 \preceq \overline{\gamma}_2.[\overline{\gamma}_1 \mapsto \overline{\tau}_1]\tau_2}$$

Let us write $\bot$ for $\gamma.\gamma$.

# Unification of shapes modulo $E$

We extend the purely syntactic unification into a unification *modulo* $E$.

We write $E \models s_1 \preceq s_2$ if there exists a shape $s$ such that $s_1 \preceq s$ and $E \models s = s_2$.

We also write $E \models s_1 = s_2$ if $E \models s_1 \preceq s_2$ and $E \models s_2 \preceq s_1$.

**Example**

$\gamma_1.\ \alpha \times \gamma_1 \preceq \text{int} \times \text{int}$ does not hold, but $\alpha = \text{int} \models \gamma_1.\ \alpha \times \gamma_1 \preceq \text{int} \times \text{int}$ does.

# Informative representant

Under a fixed set of type equalities, some representants are more informative than others. Indeed, under $\alpha = \beta_1 \to \beta_2$, if an expression is found to have both shapes $\alpha$ and $\gamma.\gamma \to \beta_2$, it is a better idea to keep propagating $\beta_1 \to \beta_2$ for the shape of this expression, because **it says more about the structure** of the evaluation of this expression than $\alpha$.

We define a normalization procedure for shapes, written $s \lfloor_E$ where:

$$\overline{\gamma}.\tau \lfloor_E \quad = \quad \overline{\gamma}.(\tau \lfloor_E)$$

and $\tau \lfloor_E$ is the normal form of the following rewriting system:

$$\begin{array}{llll} \alpha & \rightsquigarrow_E & \alpha' & \text{if } E \models \alpha = \alpha' \text{ and } \alpha < \alpha' \\ \alpha & \rightsquigarrow_E & \varepsilon\ \overline{\tau}_1\ \overline{\tau}_2 & \text{if } E \models \alpha = \varepsilon\ \overline{\tau}_1\ \overline{\tau}_2 \\ \alpha & \rightsquigarrow_E & \tau_1 \to \tau_2 & \text{if } E \models \alpha = \tau_1 \to \tau_2 \end{array}$$

for an arbitrary [1] chosen order between type variables.

[1] Is it harmful?

# Denotation

As we pointed out earlier, some precautions must be taken when we carry an annotation through expressions that are typed under different local assumptions. More precisely, a specific shape $s$ might have different denotations under differents local assumptions:

## Denotation of a shape

The denotation of a shape $s$ under $E$ is the set of all types $\tau$ such that $E \models s \preceq \tau$.

## Examples

- The denotation of $\gamma.\alpha \to \gamma$ under true is the set of types of the form $\alpha \to \tau$, whereas under $\alpha = \beta_1 \times \beta_2$, this set is extended with types of the form $\beta_1 \times \beta_2 \to \tau$.
- The denotation of $\gamma.\gamma \times \beta_2$ under $\alpha = \beta_1 \times \beta_2$ contains all type of the shape $\tau \times \beta_2$ and $\alpha$.

# Pruning

When the local type inference propagates a shape from a clause of a pattern matching to the outside, the denotation of this shape changes. For instance, if $\alpha = \beta$ holds in a particular branch, it is fine to propagate a shape $\alpha$ or a shape $\beta$ inside this branch. Yet, outside the branch is does make a difference to choose between $\alpha$ and $\beta$ because $\alpha = \beta$ no longer holds! In that case, there is no better choice of $\bot$.

## Pruning

The pruning of $s'$ with respect to $E$ and $E'$, written $s' \restriction_{E,E'}$, is the least upper bound of the shapes $s$ such that $s \preceq s'$ holds and the denotation of $s$ under $E$ contains the denotation of $s'$ under $E \wedge E'$

## Example

We have $\gamma.\alpha \to \gamma \restriction_{\mathsf{true},\alpha=\beta_1\times\beta_2} = \gamma_1.\ \gamma_1 \to \gamma_2.$

# Some notations

We write $(\overline{\gamma}_1.\tau_1) \to (\overline{\gamma}_2.\tau_2)$ for $\overline{\gamma}_1\overline{\gamma}_2.\tau_1 \to \tau_2$ (if $\overline{\gamma}_1 \# \mathbf{ftv}(\tau_2)$ and $\overline{\gamma}_2 \# \mathbf{ftv}(\tau_1)$).

We also define the domain $\mathcal{D}(.)$ of a shape:

$$\begin{aligned} \mathcal{D}(\bot) &= \bot \\ \mathcal{D}(\overline{\gamma}.\ \tau_1 \to \tau_2) &= \tau_1 \end{aligned}$$

as well as a the codomain $\mathcal{C}(.)$ of a shape:

$$\begin{aligned} \mathcal{C}(\bot) &= \bot \\ \mathcal{C}(\overline{\gamma}.\ \tau_1 \to \tau_2) &= \tau_2 \end{aligned}$$

# Bidirectional local type inference

Using the shape local unification, one can think of many algorithms to propagate user type annotations. Standard presentation of local type inference uses **bidirectional** type inference, which is characterized by two mutually recursive judgments.

One judgment describes type inference when it is done in **checking** mode. It is activated when an **expected** shape is propagated from the context of the expression to its subexpressions:

$$E, \Gamma \vdash t \Downarrow s \rightsquigarrow t'$$

The other describes type inference in **inference** mode. It is activated when a shape is computed from the subexpressions and propagated to the context:

$$E, \Gamma \vdash t \Uparrow s \rightsquigarrow t'$$

The rules defining these judgments maintain the invariant that $s$ is normalized with respect to $E$.

# A glimpse at local type inference rules : Functions

$$\textsc{Lam-}\Uparrow\quad \frac{E, \Gamma; x : \theta \lfloor_E \vdash t \Uparrow s \rightsquigarrow t'}{E, \Gamma \vdash \lambda(x : \theta).t \Uparrow (\theta \lfloor_E) \to s \rightsquigarrow \lambda(x : \theta \lfloor_E).t'}$$

$$\textsc{Lam-}\Downarrow\quad \frac{\begin{array}{c} s' = s \sqcup (\theta \lfloor_E \to \bot) \\ E, \Gamma; x : \mathcal{D}(s') \vdash t \Downarrow \mathcal{C}(s') \rightsquigarrow t' \end{array}}{E, \Gamma \vdash \lambda(x : \theta).t \Downarrow s \rightsquigarrow \lambda(x : \theta \lfloor_E).t'}$$

Lam-$\Uparrow$ injects the normalized user type annotation of the argument in the environment and it propagates the inferred shape for the body of the function to the context as the codomain of the inferred shape for the function.

Lam-$\Downarrow$ merges the information coming from the expected (normalized) shape with the user type annotation of the argument and it propagates the codomain of the resulting shape into the function body.

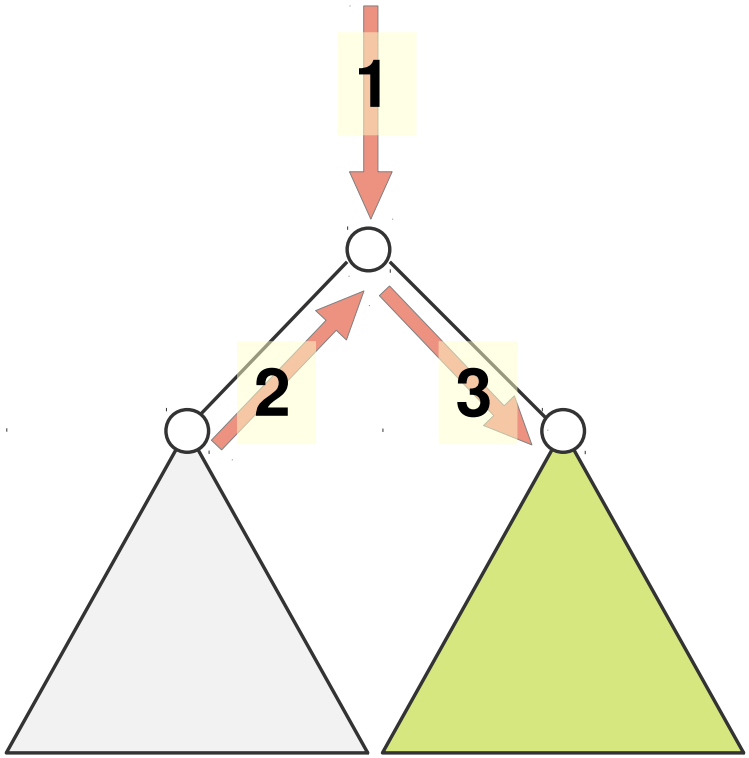
# A glimpse at local type inference rules : Applications

$$\texttt{App-}\Uparrow\quad \frac{E, \Gamma \vdash t_1 \Uparrow s \rightsquigarrow t_1' \qquad E, \Gamma \vdash t_2 \Downarrow \mathcal{D}(s) \rightsquigarrow t_2'}{E, \Gamma \vdash t_1\ t_2 \Uparrow \mathcal{C}(s) \rightsquigarrow t_1'\ t_2'}$$

$$\texttt{App-}\Downarrow\quad \frac{E, \Gamma \vdash t_1 \Uparrow s_1 \rightsquigarrow t_1' \qquad E, \Gamma \vdash t_2 \Downarrow \mathcal{D}(s_1 \sqcup (\bot \to s)) \rightsquigarrow t_2'}{E, \Gamma \vdash t_1\ t_2 \Downarrow s \rightsquigarrow t_1'\ t_2'}$$

Notice the **asymetry** between the treatments of left hand side and right hand side: the shape of $t_1$ is always inferred, even in checking mode. On the one hand, this asymetry permits a communication between $t_1$ and $t_2$, which is pleasant if $t_1$ is a variable $f$ whose type scheme is already known. On the other hand, in checking mode, the domain of the information held by the expected shape $s$ is not propagated inside $t_1$.

# Picturing local type inference



# Example 1

```
type t α =
| T1 : int → int t
| T2 : list α → α t

let apply = forall α.
  fun (x : α) p (t : α repr) →
    match t with
    | T1 n → p x ∧ n > 0
    | T2 l → (p : α → bool) (List.hd xs)
```

In the first branch, the algorithm *checks* that `p x` has type `bool`. Then, it infers that `p` has shape $\bot$, thus `x` is checked with $\bot$. As there is a type annotation $\alpha$ for `x` in the environment, we compute $\alpha \lfloor_{\alpha=\mathsf{int}}$, which is the more informative type int. To realize this conversion, a coercion is inserted from $\alpha$ to int.

# Example 1

```
type a' t =
| T1 : int → int t
| T2 : α list → α t

let apply = forall α.
  fun (x : α) p (t : α repr) →
    match t with
    | T1 n → (p : ⊥ ▷ ⊥) (x : α ▷ int) ∧ n > 0
    | T2 l → (p : α → bool) (List.hd xs)
```

The missing type annotation are reconstructed by the complete type inference of MLGX.

# Example 1

```
type α t =
| T1 : int → int t
| T2 : α list → α t

let apply = forall α.
  fun (x : α) (p : α → bool) (t : α repr) →
    match t with
    | T1 n →
        (p : ⊥ → bool ▷ int → bool) (x : α ▷ int)
        ∧ n > 0
    | T2 l → (p : α → bool) (List.hd xs)
```

# Example 2

```
type α t = I : int t

let id : forall α. α t → α → α =
  fun r x → match r with I → x
```

This program is easily handled by the local type inference.

# Example 2

```
type α t = I : int t

let id : forall α. α t → α → α =
  fun (r : α t) (x : α) → match (r : α t) with I →
    ((x : α ▷ int) : int ▷ α)
```

One coercion is inserted to normalize the shape of `x` with respect to $\alpha = \text{int}$ and another one to convert the shape of `x` inside the branch to the expected shape coming from the return type of the function.

# Example 2

```
type α t = I : int t

let id : forall α. α t → α → α =
  fun (r : α t) (x : α) →
    (fun y →
      match (r : α t) with I →
        ((x : α ▷ int) : int ▷ ⊥)
    ) y
```

Again, MLGX complete type inference is able to fill the remaining holes.

# Example 2

```
type α t = I : int t

let id : forall α. α t → α → α =
  fun r x → (fun y → match r with I → x) y
```

Now assume that a simple program transformation like $\eta$-expansion is applied to our program. The expected shape $\alpha$ that comes from the return type of `id` is not propagated to the occurrence of `x` inside the pattern matching. It would be unsound to propagate directly the shape int that is inferred for `x` at that point, that is why int is pruned, which amounts to insert a coercion $(\texttt{x} : \text{int} \triangleright \bot)$ in the branch of the pattern matching.

# A fragile design choice

The asymmetric rules for application behave badly in presence of anonymous functions. Assume a data constructor `Int` with type `int repr`, that makes the equation $\alpha$ = `int` available inside the body of the branch in the following anonymous function:

```
let rec double : forall α. α repr → α list → α list =
  fun t l →
      map (fun x → match t with I → x + x) l
```

The type of `l` is given by the user type annotation. Yet, bidirectional local type inference is uncapable of propagating this type from `l` to the term

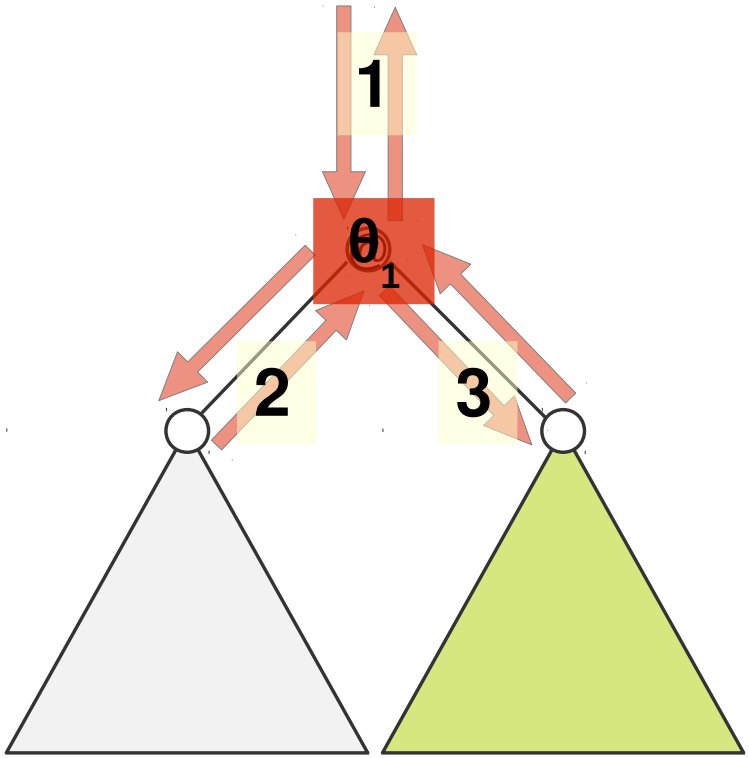
```
map (fun x → match t with I → x + x)
```

# Iterated local type inference

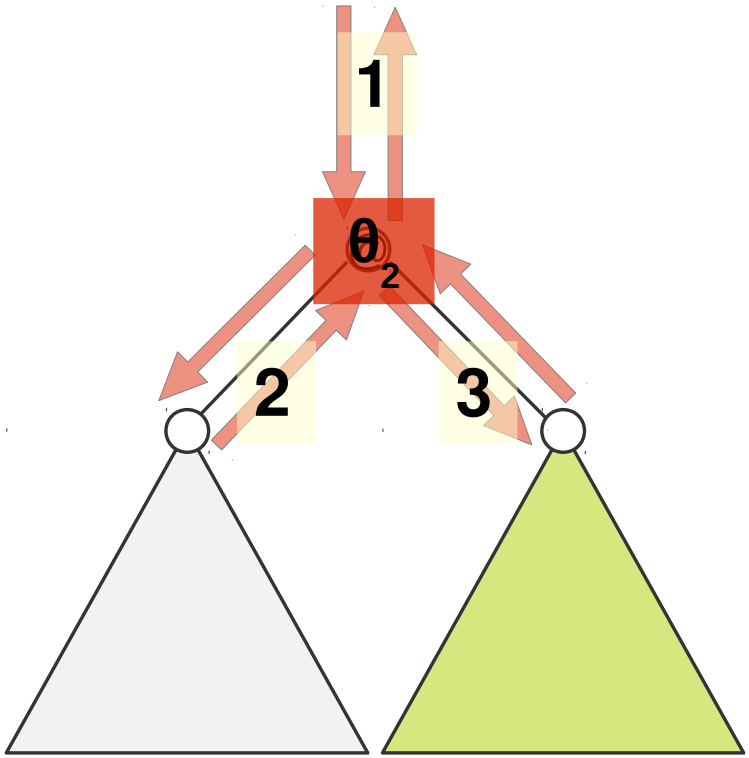
Another local type inference algorithm is presented in the paper [Pottier and Régis-Gianas, 2006]. It is:

- Simultaneously Bidirectional:
  It conveys an inferred shape as well as an expected shape.
- Iterated:
  Its rules can be applied several times to improve the propagation of type annotations.
- Symmetric:
  It uses a symmetric application rule that propagates the inferred shape of the previous iteration for the left-hand side of the application as an expected shape for the right-hand side of the application, and vice et versa.

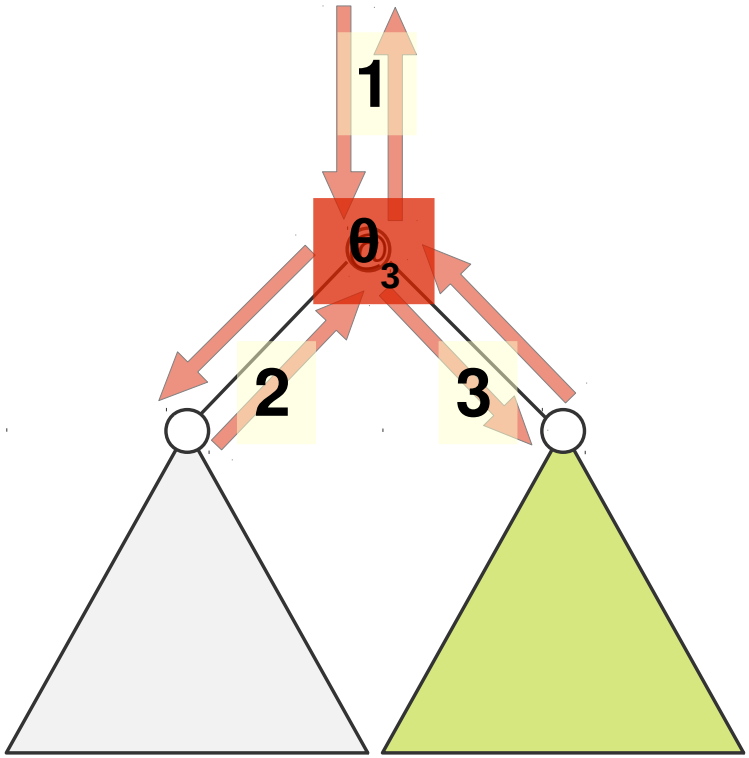
# Picturing local type inference



# Picturing local type inference

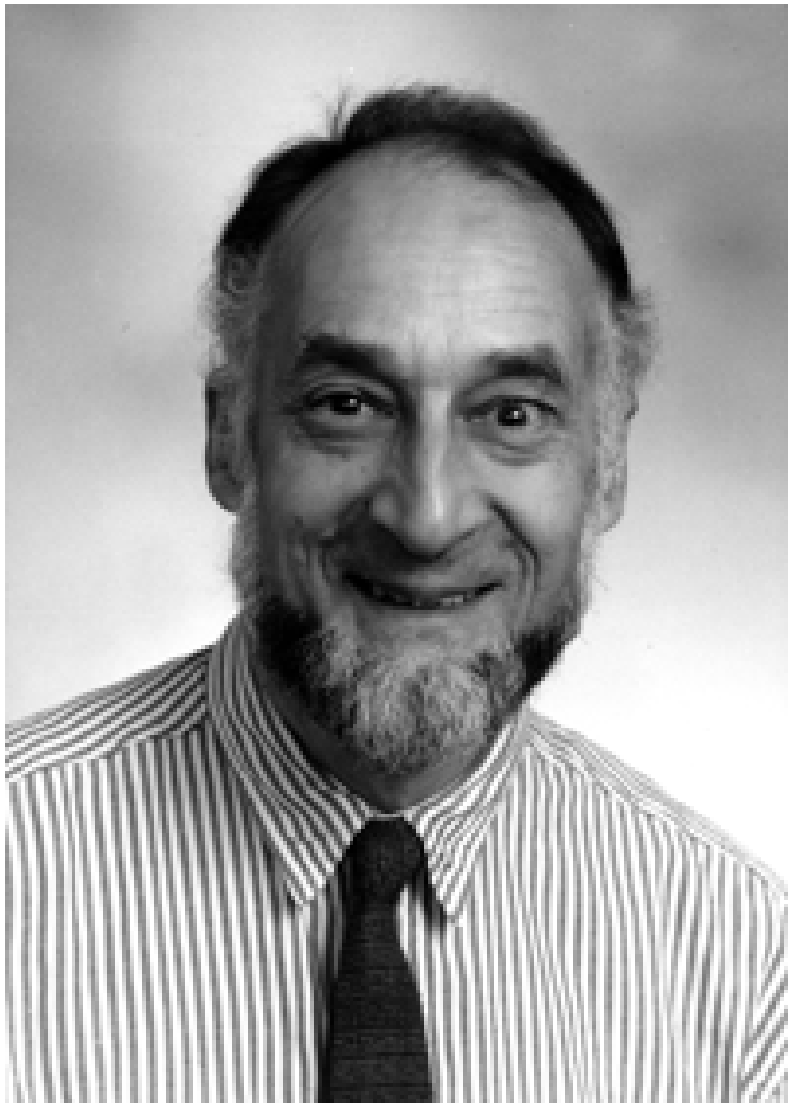


# Picturing local type inference



# Contents

# 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.

# Example (inspired by [Vytiniotis et al., 2011])

```
let fr : forall α. α → α 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?

# Example (inspired by [Vytiniotis et al., 2011])

```
let fr : forall α. α → α repr → bool =
  fun x r →
    let g : ∀β[α = bool].β → 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?

# 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!

```
let fr : forall α. α → α repr → bool =
  fun x r →
    match r with
    | TyBool → fun z → not x
    | _ → true
```

# No! This program should be rejected.

```
let fr : forall α. α → α repr → bool =
  fun x r →
    let g = fun z → not x in
    match r with
    | TyBool → g ()
    | _ → 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

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

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

# 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:

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

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

The constraint generation rule for annotated `let` is:

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

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

# Example

```
let fr =
  fun x r →
    let g = fun z → not x in
    match r with
    | TyBool → g ()
    | _ → true
```

The standard typing constraint for this program looks like:

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

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

# Example

```
let fr =
  fun x r →
    let g = fun z → not x in
    match r with
    | TyBool → g ()
    | _ → true
```

When generalization is ruled out, typing constraint looks like:

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

In that case, $\gamma_x$ must be unified with bool.

# Example

```
let fr =
  fun x r →
    let g = fun z → not x in
    match r with
    | TyBool → g ()
    | _ → true
```

After some simplifications, we have:

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

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

# Example

```
let fr =
  fun x r →
    let g = fun z → not x in
    match r with
    | TyBool → g ()
    | _ → true
```

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

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

...

# 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 $\phi$, all the free type variables that are not assigned by $\phi$ 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 [2] 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.

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

# Type-level programming

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

type family add : $\star \to \star \to \star$
type add zero $\alpha = \alpha$
type add (succ $\alpha$) $\beta$ = succ (add $\alpha$ $\beta$)

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...

# Bibliography I


- Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 252–265, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: http://doi.acm.org/10.1145/268946.268967.

François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 232–244. ACM, 2006. ISBN 1-59593-027-2.

- Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for gadts. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 341–352, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596599.

Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, December 2005. To appear.

# Bibliography II

Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, 2011.

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