Stratified Type Inference for Generalized Algebraic Data Types

François Pottier and Yann Régis-Gianas

POPL 2006, Charleston
January 13, 2006

What are GADTs?

- With GADTs, data constructors can have more precise type schemes than with usual algebraic data types.
- Here is a GADT declaration for the abstract syntax of a simply typed toy language:

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{term} :: \star \to \star = & & \\
\quad | \quad \mathit{Lit} & : & \mathit{int} \to \mathit{term}\ \mathit{int} \\
\quad | \quad \mathit{Inc} & : & \mathit{term}\ \mathit{int} \to \mathit{term}\ \mathit{int} \\
\quad | \quad \mathit{Pair} & : & \forall \gamma \gamma'.\mathit{term}\ \gamma \times \mathit{term}\ \gamma' \to \mathit{term}\ (\gamma \times \gamma') \\
\quad | \quad \mathit{Fst} & : & \forall \gamma \gamma'.\mathit{term}\ (\gamma \times \gamma') \to \mathit{term}\ \gamma \\
\quad | \quad \mathit{Snd} & : & \forall \gamma \gamma'.\mathit{term}\ (\gamma \times \gamma') \to \mathit{term}\ \gamma'
\end{array}
$$

GADTs give rise to local equations

This is a tagless interpreter for terms:

$$
\begin{array}{l}
\mathsf{fix}\,(\mathit{eval} : \forall\alpha.\mathit{term}\ \alpha \to \alpha).\lambda t. \\
\quad \mathsf{case}\ t\ \mathsf{of} \\
\quad\quad \mid \mathit{Inc}\ t' \to (*\ \mathit{Inc} : \mathit{term}\ \mathit{int} \to \mathit{term}\ \mathit{int}\ \ \mathit{hence}\ \alpha = \mathit{int}\ *) \\
\quad\quad\quad \mathit{eval}\ t' + 1 \\
\quad\quad \mid \ldots
\end{array}
$$

The problem of type inference

Roughly speaking, what makes type inference for GADTs difficult is to answer these questions:

- What local equations are available?
- Where and how are they used?

The problem of type inference

- What local equations are available?

$$\mathbf{type}\ \mathit{eq}\ ::\ \star \to \star \to \star =$$
$$\mathit{Eq} : \forall \alpha.\mathit{eq}\ \alpha\ \alpha$$

$$\text{let}\ \mathit{cast}\ t\ x = \text{case}\ t\ \text{of}\ \mathit{Eq} \to x$$

- Assuming $t : \mathit{eq}\ \tau_1\ \tau_2$, the local equation is $\tau_1 = \tau_2$. Yet, during type inference, τ_1 and τ_2 are unknown, and so is the equation ...

The problem of type inference

- Where and how are they used?

$$\begin{array}{l}\mathbf{type}\ \mathit{eq}\ ::\ \star \to \star \to \star = \\ \quad \mathit{Eq} : \forall \alpha.\mathit{eq}\ \alpha\ \alpha\end{array}$$

$$\text{let } \mathit{cast}\ t\ x = \text{case } t \text{ of } \mathit{Eq} \to x$$

- Assuming $t : \mathit{eq}\ \tau_1\ \tau_2$ and $x : \tau_1$, then the local equation is $\tau_1 = \tau_2$ and the return type can be:
  - τ_2 by using the local equation.
  - τ_1 by not using the local equation.
- Finding out about all choices requires knowing which local equations are available.

The problem of type inference

- The existence of choices means that there is no principal type:

$$\mathbf{type}\ \mathit{eq}\ ::\ \star \to \star \to \star =$$
$$\mathit{Eq} : \forall \alpha.\mathit{eq}\ \alpha\ \alpha$$

$$\text{let}\ \mathit{cast}\ t\ x = \text{case}\ t\ \text{of}\ \mathit{Eq} \to x$$

- These two ML type schemes are valid for *cast* and are incomparable:
  - $\forall \gamma\gamma'\gamma''.\mathit{eq}\ \gamma'\ \gamma'' \to \gamma \to \gamma$
  - $\forall \gamma\gamma'.\mathit{eq}\ \gamma\ \gamma' \to \gamma \to \gamma'$
- Thus, user annotations seem unavoidable.

Is life simpler in everyday programs?

With just a few type annotations, type inference becomes very much like type checking:

$$
\begin{array}{l}
\mathsf{fix}\,(\mathit{eval} : \forall\alpha.\mathit{term}\ \alpha \to \alpha).\lambda t. \\
\quad \mathsf{case}\ t\ \mathsf{of} \\
\qquad \mid \mathit{Inc}\ t \to (*\ \alpha = \mathit{int}\ *) \\
\qquad\quad \mathit{eval}\ t + 1 \\
\qquad \mid \ldots
\end{array}
$$

Stratified type inference

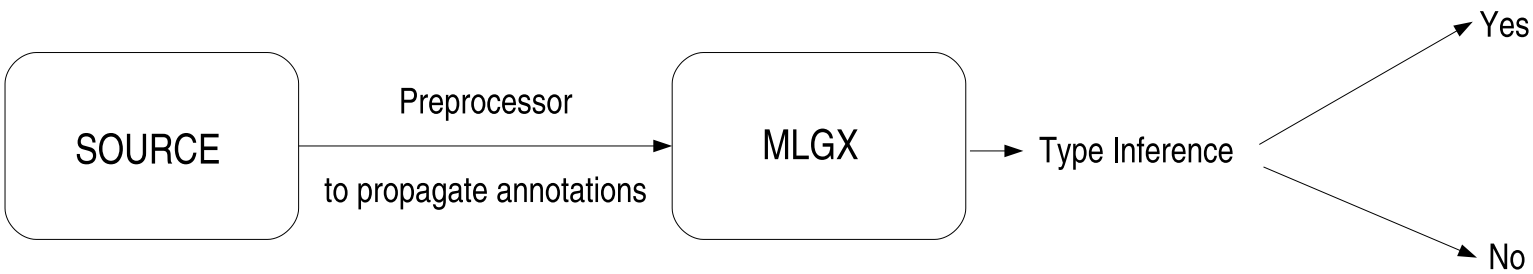
Two natural approaches:

- Implement an incomplete type inference engine dealing with ML and GADTs simultaneously (Sulzmann *et al*)

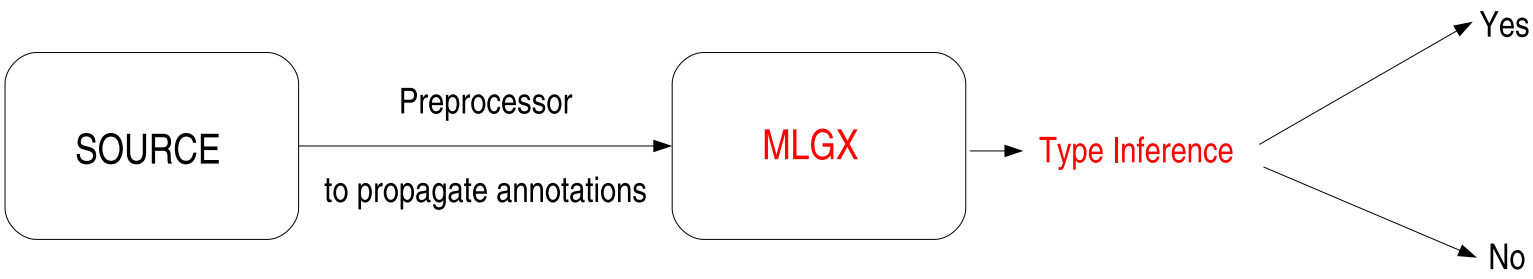
- or (the approach we will describe today):
  1. Design a restricted core language where enough annotations are required to make complete type inference as easy as in ML ;
  2. Design an incomplete but predictable algorithm to propagate user annotations.

  This approach is followed by Peyton Jones *et al*, except they do not clearly separate 1 and 2.

Stratified type inference at a glance



Stratified type inference at a glance



MLGX: ML with explicit typing of GADTs

First restriction:

Case constructs must be annotated.

$$\mathsf{case}\,(t : \tau)\,\mathsf{of}\ \ldots$$

- Only this annotation will be used to deduce the local equations, not the inferred type. Thus, local equations are known everywhere even before type inference begins.

MLGX: ML with explicit typing of GADTs

Second restriction:

An explicit coercion is required to use an equation.

$$(t : \tau_1 \triangleright \tau_2)$$

- This coercion is valid only if $\tau_1 = \tau_2$ is implied by the local equations.
- This implication test is possible because the local equations are known everywhere.
- Once this test is done, this coercion can be viewed as an application of the identity at type $\tau_1 \longrightarrow \tau_2$ to t.

Properties of MLGX

- In MLGX, type inference is simple:
  1. Every coercion is checked.
  2. Viewing coercions as constants, standard ML type inference is performed.
- MLGX has principal types.
- Every ML program is well-typed in MLGX.

Our example in MLGX

- Our example must be annotated to be well-typed in MLGX:

$$
\begin{array}{l}
\mathsf{fix}\,(\mathit{eval} : \forall\alpha.\mathit{term}\ \alpha \to \alpha).\lambda t. \\
\quad \mathsf{case}\,(t : \mathit{term}\ \alpha)\ \mathsf{of} \\
\qquad \mid \mathit{Inc}\ t \to (\mathit{eval}\ t + 1 : (\mathit{int} \triangleright \alpha)) \\
\qquad \mid \ldots
\end{array}
$$

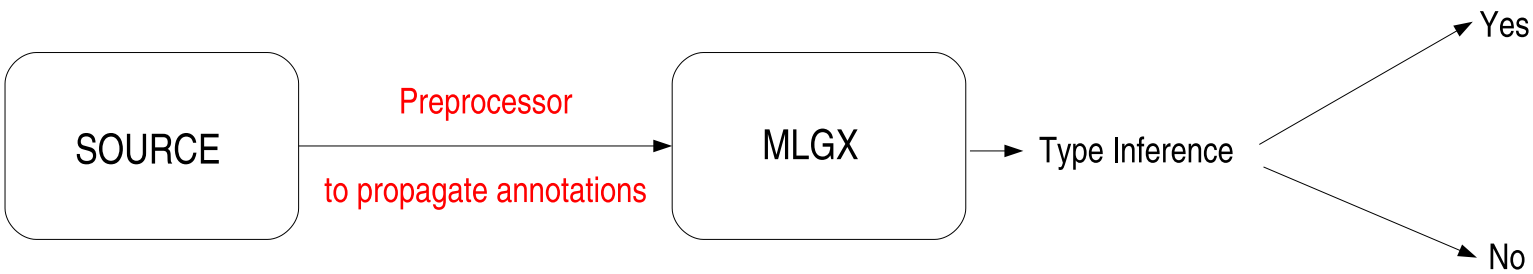
Our example in MLGX

- Our example must be annotated to be well-typed in MLGX:

$$\begin{array}{l}\mathsf{fix}\,(\mathit{eval} : \forall\alpha.\mathit{term}\ \alpha \to \alpha).\lambda t.\\ \quad \mathsf{case}\,(t : \mathit{term}\ \alpha)\ \mathsf{of}\\ \quad\quad \mid \mathit{Inc}\ t \to (\mathit{eval}\ t + 1 : (\mathit{int} \triangleright \alpha))\\ \quad\quad \mid \ldots\end{array}$$

- The red annotations are hard to infer whereas it seems easy to deduce the blue ones from the red ones.
  $\Rightarrow$ Why not do so automatically?

Stratified type inference at a glance



Local shape inference as a preprocessor

- The idea is to design a preprocessor to deduce the blue annotations from the red ones.
- This preprocessor will necessarily be incomplete, hence it should be predictable.
- How to achieve predictability?
  $\Rightarrow$ By preventing long-distance unification.
  $\Rightarrow$ We introduce an algebra of shapes to support local propagation of information.

Shapes

- A shape looks exactly like a type scheme, but is interpreted as a type approximation.
- The shape $\gamma.\gamma \to \gamma$ describes both the identity function and the integer successor function.
- Shapes are closed. An unknown is never shared between two shapes. This prevents long-distance unification.
- Shapes can be combined by first order unification:

$$(\gamma.\gamma \to \gamma) \sqcup (\gamma'.\gamma' \to \mathit{int}) = \mathit{int} \to \mathit{int}$$

Shape inference in action

- Shape inference is rigorously defined in the paper.
- Shape inference is able to deduce the type of t, to deduce the type of the case construct, and to insert the necessary annotation and coercion:

$$
\begin{array}{l}
\mathsf{fix}\,(\mathit{eval} : \forall\alpha.\mathit{term}\ \alpha \to \alpha).\lambda t. \\
\quad \mathsf{case}\,(t : \mathit{term}\ \alpha)\ \mathsf{of} \\
\qquad \mid \mathit{Inc}\ t \to (\mathit{eval}\ t + 1 : (\mathit{int} \triangleright \alpha)) \\
\qquad \mid \ldots
\end{array}
$$

Shape inference in action

- Thanks to shape unification, local shape inference is robust in the face of minor changes:

$$\begin{array}{l}\mathsf{fix}\,(\mathit{eval} : \forall\alpha.\mathit{term}\ \alpha \to \alpha).\lambda t.\\ \quad \mathsf{case}\,(\mathsf{id}\ \mathsf{t} : \mathit{term}\ \alpha)\ \mathsf{of}\\ \quad\quad \mid \mathit{Inc}\ t \to (\mathit{eval}\ t + 1 : (\mathit{int} \triangleright \alpha))\\ \quad\quad \mid \ldots\end{array}$$

- Indeed, $(\gamma.\gamma \to \gamma) \sqcup (\gamma'.\mathit{term}\ \alpha \to \gamma')$ is $\mathit{term}\ \alpha \to \mathit{term}\ \alpha$.

Summary about MLGX

- MLGX is a simple extension of ML that introduces GADTs while preserving principal types.
- It is robust in the sense that there are not many ways of altering its design.

Summary about local inference

- Stratified type inference enables a modular extension of ML type inference. Because local shape inference is incomplete. it is natural to keep it apart from MLGX.
- We have experimented with two different shape inference algorithms based on the same shape toolbox. One is inspired from GHC's wobbly type inference. The other is strictly more precise.
- Both algorithms are sound in a sense defined in the paper.
- This is an original use of local inference in a language *à la* ML (as opposed to System F).