Simple Unification-based Type Inference for GADTs

Stephanie Weirich
University of Pennsylvania

joint work with Dimitrios Vytiniotis, Simon Peyton Jones and Geoffrey Washburn
Overview

• Goal: Add GADTs to Haskell
• Problem: GADT type inference is hard
• Requirements:
  – Simple, declarative specification
  – Easy to implement in GHC
• Solution: Type annotations
• Non-goal: type as many programs as possible
A typical evaluator

data Term = Lit Int
           | Succ Term
           | IsZero Term
           | If Term Term Term

data Value = VInt Int | VBool Bool

eval :: Term -> Value
eval (Lit i) = VInt i
eval (Succ t) = case eval t of { VInt i -> VInt (i+1) }
 eval (IsZero t) = case eval t of { VInt i -> VBool (i==0) }
 eval (If b t1 t2) = case eval b of
                     VBool True -> eval t1
                     VBool False -> eval t2
data Term a where
  Lit :: Int -> Term Int
  Succ :: Term Int -> Term Int
  IsZero :: Term Int -> Term Bool
  If :: Term Bool -> Term a -> Term a -> Term a

eval :: Term a -> a
eval (Lit i) = i
eval (Succ t) = 1 + eval t
eval (IsZero i) = eval i == 0
eval (If b e1 e2) = if eval b then eval e1 else eval e2

• In a case alternative, we learn more about ‘a’; we call this type refinement
• Can’t construct ill-typed terms: (If (Lit 3) ...)
• Evaluator is simpler and more efficient
Algebraic Data Types

Normal Haskell or ML data types:

```
data T a = T1 | T2 Bool | T3 a a
```

gives rise to constructors with types

```
T1 :: T a  
T2 :: Bool -> T a  
T3 :: a -> a -> T a
```

Return type is always (T a)
Generalized Algebraic Data Types

- Allow arbitrary arguments to return type
- Programmer gives types of constructors explicitly
- Subsumes standard algebraic datatypes and datatypes with “existential components” [LO94]

```haskell
data Term a where
    Lit     :: Int -> Term Int
    Succ    :: Term Int -> Term Int
    IsZero  :: Term Int -> Term Bool
    If      :: Term Bool -> Term a -> Term a -> Term a
```
GADTs have MANY applications

- Language description and implementation (Typed evaluators, Pugs)
- Domain-specific embedded languages (Darcs, Yampa)
- Generic programming (Hinze et al., Weirich)
- “Dependent” types (Xi et al., Sheard)
Adding GADTs to GHC

- Simple extension
  - No big changes to language semantics
  - No re-engineering compiler
- GADTs generalize algebraic datatypes
  - Uniform mechanism for ADTs, “existential components” and GADTs
  - All existing Haskell programs still work
- Complete specification of type inference
  - Predictability
Yes....

- Construction is simple: constructors are just ordinary polymorphic functions
- All the constructors are still declared in one place
- Pattern matching is still strictly based on the value of the constructor; the dynamic semantics is type-erasing
Complete type inference is hard

- Many examples require polymorphic recursion
- Even for those that don’t, problems remain

```haskell
data T a where
  C :: Int -> T Int

f (C x) = 3 + x
```

- What is type of \( f \) ?

```
T Int -> Int
∀ a. T a -> Int
∀ a. T a -> a
```

Neither type is more general
Annotations solve the problem

- Naïve specification allows multiple types for f
  - T Int -> Int
  - ∀ a. T a -> Int
  - ∀ a. T a -> a

- Inference algorithm can assign just one

- Annotations remove this ambiguity *in the specification*
Basic idea

- Type system distinguishes between *inferred* types and those *known* from user annotations
- Typing context and judgment tracks when types are *wobbly* or *rigid*
  
  **Modifiers**  
  \[ m, n ::= w | r \]

  **Environments**  
  \[ \Gamma, \Delta ::= . | \Gamma, x:^m \sigma \]

  **Judgment**  
  \[ \Gamma |- t :^m \tau \]

- \[ \Gamma |- t :^r \tau \] means that term \( t \) has type \( \tau \) when we know \( \tau \) *completely in advance*.
- \[ \Gamma |- t :^w \tau \] checks without that assumption.
GADT refinement

- GADT refinement only involves rigid type information
  - Rigid scrutinee triggers refinement
  - Only rigid context types refined
  - Only rigid result types refined

- Example:

```haskell
data T a where C :: Int -> T Int
f :: forall a.T a -> a -> Int
f (C x) y = x + y
```

-- inferred type: T Int -> Int -> Int
```haskell
g (C x) y = x + y
```
Type checking a case expression

\[
\frac{
  \begin{align*}
    x &: m \, \tau_p \in \Gamma \\
    \Gamma \vdash p \to t : <m,n> \, \tau_p \to \tau_t
  \end{align*}
}{
  \Gamma \vdash \text{(case } x \text{ of } p \to t) : n \, \tau_t
}\]
Type checking a wobbly branch

Guess instantiation of type arguments to \( C \)

\[
\begin{align*}
C : \forall a. \overline{\tau_1} \rightarrow \overline{\tau_2} & \in \Gamma \quad a \notin \text{fv}(\Gamma, \overline{\tau_p}, \overline{\tau_t}) \\
\overline{a_c} & = \overline{a} \cap \text{fv}(\overline{\tau_2}) \\
\theta & = \left[a_c \mapsto \overline{v}\right] \\
\theta(\overline{\tau_2}) & = \overline{\tau_p} \\
\Gamma, x : w \theta(\overline{\tau_1}) & \vdash t :^m \tau_t \\
\hline
\Gamma \vdash C \overline{x} \rightarrow t^{(w, m)} \quad \text{T} \overline{\tau_p} \rightarrow \tau_t
\end{align*}
\]

PCON-W

Type of scrutinee is wobbly

data \text{T} a \text{ where } C :: a \rightarrow \text{T} a \\
f y = \text{case } y \text{ of } C x \rightarrow x + 1
Type checking a rigid branch

Refine **rigid** parts of the context

Unify scrutinee type with result type of constructor

Refine result type if it is rigid

\[
\begin{align*}
C :^r \forall \bar{\tau}_1 \rightarrow \bar{T} \bar{\tau}_2 \in \Gamma & \quad \text{Unify ftv}(\Gamma, \bar{\tau}_1, \bar{\tau}_t) \\
\theta \in \text{fmgu}(\bar{\tau}_p \equiv \bar{\tau}_2) & \\
\theta(\Gamma, x : ^r \bar{\tau}_1) \vdash t : ^m \theta^m(\bar{\tau}_t) & \\
\hline
\Gamma \vdash C \bar{x} \rightarrow t : ^{r,m} \bar{T} \bar{\tau}_p \rightarrow \bar{\tau}_t & \text{PCON-R}
\end{align*}
\]

```haskell
data T a b where C :: b -> T Int b  
f :: T a Bool -> a -> a  
f y w = case y of C x -> if x then w else 1
```
Checking + inference is hard

- MGU is the standard way to solve constraints and produce a substitution
  - Used in Algorithm W
- We really thought this would work
- But, even with all these annotations
  - scrutinee of case
  - return type of case
  - all refined variables in context

MGU does not produce a complete specification of what programs typecheck.
Pathological example

Should this program type check?

data Eq a b where
  Refl :: Eq c c

f :: ∀a b. Eq a b -> (a -> Int) -> b -> Int
f x y z = (\w -> case x of Refl -> y w) z

Context: \( x^r \) Eq a b, \( y^r \) a->Int, \( z^r \) b, \( w^w \) b

Compute \( \theta = \text{MGU}(\text{Eq } c \ c = \text{Eq } a \ b) \)

If \( \theta = \{ a \Rightarrow b, c \Rightarrow b \} \) then yes
If \( \theta = \{ b \Rightarrow a, c \Rightarrow a \} \) then no
If \( \theta = \{ b \Rightarrow c, a \Rightarrow c \} \) then no
“Fresh” - mgu

- Problem
  - Choice of MGU makes refined type match type of wobbly variable

- A solution
  - Choose the right(?) MGU

- Our solution
  - Don’t choose any MGU
  - In this situation, never let refined type match wobbly type
  - Choose a fresh variable \( d \) and use unifier
    \[ \theta = \{ a \Rightarrow d, b \Rightarrow d, c \Rightarrow d \} \]
  - Rejects pathological example
Other details

• Additional rules to locally propagate type annotations
  – Like shape inference pass (Pottier/Régis-Gianis)

• Lexically-scoped type variables
  – Must be able to annotate all sub-expressions
  – Bind both “universally” and “existentially” quantified type variables in program text

• Nested patterns
  – more complicated rules, but straightforward

• See paper for details
Non-monotonic annotations

- Sometimes adding an annotation can cause a working program to be rejected
- Hasn’t been a problem in practice
- Pathological example:

```haskell
data T a where C :: T Int
x :r T a, y :w a ⊢ case x of C -> y :w a

- Making \( y \) rigid triggers refinement

```haskell
x :r T a, y :r a ⊢ case x of C -> y :w a
```

- Making return type rigid restores typability

```haskell
x :r T a, y :r a ⊢ case x of C -> y :r a
```
Formal Properties

• Type system is *sound*
  – Type-preserving translation to explicitly-typed language (System F + GADT)
  – Soundness proved for explicit language

• Type system is *expressive*
  – Any program in explicitly-typed language acceptable (with enough annotations)

• Type inference algorithm is *sound* and *complete*

• Type system is a *conservative extension* of Hindley-Milner system

• All details in companion technical report
Related work

• Pottier and Simonet
  – Use implication constraints for complete type inference
  – Solving such constraints can be intractable

• Our previous (unpublished) version
  – Implemented in a previous version of GHC
  – More complicated than this system: types may be partly wobbly

• Pottier and Régis-Gianis
  – Constraint-based
  – Shape inference pass to propagate local annotations
  – Second pass for explicit type system

• Sulzmann et al.
  – Constraint-based
  – Abandons complete type inference
  – Concentrates on error messages
Future work

• Resolve poor interaction between type classes and GADTs

• More general way to combine type inference and type checking
  – slightly different mechanism useful for higher-rank/impredicative polymorphism

• Towards dependently-typed programming languages
Conclusions

• GADTs implemented in GHC
• Can extend unification-based type inference with GADTs
• Simple specification of type system due to user annotations
• Complete specification of where type annotations are necessary is important