Complete and decidable type inference for GADTs

Tom Schrijvers

Katholieke Universiteit Leuven, Belgium
TOM.SCHRIJVERS@CS.KULEUVEN.BE

Brouwer Institute, Nijmegen – March 24, 2009
Overview

1. GADTs
   - Overview
   - Type Checking
2. The OutsideIn Algorithm
3. Formal Specification and Properties
4. Related Work
5. Summary

Type inference for GADTs

Tom Schrijvers
1. GADTs
   - Overview
   - Type Checking
2. The OutsideIn Algorithm
3. Formal Specification and Properties
4. Related Work
5. Summary
(Plain) Algebraic Data Types

**Expression Type**

```haskell
data Exp = VAL Int     -- n
    | GT Exp Exp     -- e > e
    | IF Exp Exp Exp -- if e1 then e2 else e3

good, bad :: Exp

good = IF (VAL 7 'GT' VAL 3) (VAL 42) (VAL 3)

bad = IF (VAL 7) (VAL 42) (VAL 3)
```
Expression Type

data Exp a where
  VAL :: Int -> Exp Int
  GT :: Exp Int -> Exp Int -> Exp Bool
  IF :: Exp Bool -> Exp a -> Exp a -> Exp a

  good :: Exp Int
  good = IF (VAL 7 'GT' VAL 3) (VAL 42) (VAL 3)

-- bad is rejected by the type checker
-- bad :: Exp Int
-- bad = IF (VAL 7) (VAL 42) (VAL 3)
GADTs, alternative syntax

Expression Type

```haskell
data Exp a
  = a ~ Int => VAL Int
  | a ~ Bool => GT (Exp Int) (Exp Int)
  | IF (Exp Bool) (Exp a) (Exp a)
```

- equality constraints a ~ Int and a ~ Bool
- a.k.a. refinements
Using GADTs

Expression Type

data Exp a
  = a ~ Int => VAL Int
  | a ~ Bool => GT (Exp Int) (Exp Int)
  | IF (Exp Bool) (Exp a) (Exp a)

eval :: Exp a -> a
eval (VAL n) = n
eval (GT e1 e2) = eval e1 > eval e2
eval (IF e1 e2 e3) = if eval e1
  then eval e2
  else eval e3
GADT Applications

- typed interpreter
- typed type-preserving compiler
- typed parsers
- darcs theory of patches
- typed EDSLs
- length-indexed lists
- ...

Type inference for GADTs
Type Checker

- **type checking**

  *Does the code have the type promised by the signature?*
  
given both code and signature
Type Checker

- **type checking**

  *Does the code have the type promised by the signature?*

  given both code and signature

- **type inference**

  *What is the signature?*

  given only the code
Consider

data Exp a = a ~ Int => VAL Int | ... 

eval :: Exp a -> a 
eval (VAL n) = n 
... 

▶ Is VAL n LHS of type Exp a?
Consider

data Exp a = a ~ Int => VAL Int | ...

eval :: Exp a -> a
eval (VAL n) = n
...

▶ Is VAL n LHS of type Exp a?  Yes!
Consider

data Exp a = a ~ Int => VAL Int | ...

eval :: Exp a -> a
eval (VAL n) = n
...

- Is VAL n LHS of type Exp a? Yes!
- Is n in RHS of type a?
Consider

data Exp a = a ~ Int => VAL Int | ...

eval :: Exp a -> a
eval (VAL n) = n
...

- Is \text{VAL} \ n \ LHS \ of \ type \ Exp \ a? \hspace{1cm} \text{Yes!}
- Is \ n \ in \ RHS \ of \ type \ a? \hspace{1cm} \text{No, Int!}
Consider

\[
data \text{ Exp } a = a \sim \text{ Int } \Rightarrow \text{ VAL } \text{ Int } \mid \ldots
\]

\[
eval :: \text{ Exp } a \to a
\]

\[
eval (\text{ VAL } n) = n
\]

\[
\ldots
\]

▶ Is \text{ VAL } n \text{ LHS of type Exp } a? 

Yes!

▶ Is n in RHS of type a?

No, \text{ Int}!

▶ Hang on, we know \text{ a } \sim \text{ Int} in the scope of the \text{ VAL } n pattern match.
Consider

```haskell
data Exp a = a ~ Int => VAL Int | ... 

eval :: Exp a -> a 
eval (VAL n) = n 
... 
```

- Is \( \text{VAL } n \) LHS of type \( \text{Exp } a \)?  
  Yes!
- Is \( n \) in RHS of type \( a \)?  
  No, \( \text{Int} \)!
- Hang on, we know \( a ~ \text{Int} \) in the scope of the \( \text{VAL } n \) pattern match.  
  So yes!
Formalization of Type Checking

data Exp a = a \sim \text{Int} \Rightarrow \text{VAL Int} \mid \ldots

eval :: \text{Exp} \ a \rightarrow a

eval (\text{VAL} \ n) = n
\ldots

Implication constraints capture scope of locally given equalities:

\[ \vdash t_x \sim \text{Exp} \ a \land (a \sim \text{Int} \Rightarrow \text{Int} \sim a) \]
The type checker needs to prove type equalities (signature equals code type),
called *wanted* equalities.
The type checker needs to prove type equalities (signature equals code type),
- called \textit{wanted} equalities.

\textbf{Hindley-Milner:}
- only proof: syntactic equality, e.g., $\text{Int} \sim \text{Int}$
The type checker needs to prove type equalities (signature equals code type),
called *wanted* equalities.

Hindley-Milner:
- only proof: syntactic equality, e.g., \(\text{Int} \sim \text{Int}\)

GADTs add a new means to prove equality:
- GADT pattern matches bring equalities into scope, e.g., \(a \sim \text{Int}\),
called (locally) *given* equalities.
Type Checking Algorithm

Not particularly exciting: it works!

- push signature information inwards in the code
- use given equalities as substitution on the wanted equalities

\[
\vdash (a \sim \text{Int} \Rightarrow \text{Int} \sim a)
\]
Type Checking Algorithm

Not particularly exciting: it works!

- push signature information inwards in the code
- use given equalities as substitution on the wanted equalities

\[ \Gamma \vdash (a \sim \text{Int} \Rightarrow \text{Int} \sim a) \]

becomes:

\[ a \sim \text{Int} \vdash \text{Int} \sim a \]
Type Checking Algorithm

Not particularly exciting: it works!

- push signature information inwards in the code
- use given equalities as substitution on the wanted equalities

\[ \vdash (a \sim \text{Int} \Rightarrow \text{Int} \sim a) \]

becomes:

\[ a \sim \text{Int} \vdash \text{Int} \sim a \]

becomes after substitution \( \{\text{Int}/a\} \):

\[ \vdash \text{Int} \sim \text{Int} \]

- sound and complete
What is the signature?

given only the code:

data Exp a = a ~ Int => VAL Int | ...  
foo (VAL n) = n
What is the signature?

given only the code:

data Exp a = a ~ Int => VAL Int | ...

foo (VAL n) = n

▷ foo :: Exp a -> a
What is the signature?

given only the code:

```haskell
data Exp a = a ~ Int => VAL Int | ...

foo (VAL n) = n
```

- `foo :: Exp a -> a`
- `foo :: Exp Int -> Int`
What is the signature?

given only the code:

```haskell
data Exp a = a ~ Int => VAL Int | ...
foo (VAL n) = n
```

- `foo :: Exp a -> a`
- `foo :: Exp Int -> Int` less general
What is the signature?

given only the code:

data Exp a = a \sim\ Int \Rightarrow VAL\ Int \mid \ldots

foo (VAL\ n) = n

\begin{itemize}
  \item foo :: Exp\ a \rightarrow a
  \item foo :: Exp\ Int \rightarrow Int
  \item foo :: Exp\ a \rightarrow Int
\end{itemize}

less general, incomparable!
What signature should we get?
What signature should we get?

- the most general one!
What signature should we get?

▶ the most general one!

Hindley Milner:

▶ \textbf{principal types} property

▶ i.e. if there is a type, there is a unique most general type
What signature should we get?
  ▶ the most general one!

Hindley Milner:
  ▶ **principal types** property
  ▶ i.e. if there is a type, there is a unique most general type

GADTs:
  ▶ multiple maximal typings possible
  ▶ arbitrary choice?
What signature should we get?

- the most general one!

Hindley Milner:

- principal types property
- i.e. if there is a type, there is a unique most general type

GADTs:

- multiple maximal typings possible
- arbitrary choice?
- reject ambiguous programs
What is the type of:

data Erk x y z where
  K :: x -> y -> Erk x y Char

f (K x y) = x:y
What is the type of:

data Erk x y z where
   K :: x -> y -> Erk x y Char

f (K x y) = x:y

▶ f :: Erk a [a] c -> [a]
What is the type of:

```haskell
data Erk x y z where
  K :: x -> y -> Erk x y Char

f (K x y) = x:y
```

- $f :: Erk \ a\ [a] \ c \rightarrow [a]$
- $f :: Erk \ a\ [Char] \ a \rightarrow [Char]$
What is the type of:

```haskell
data Erk x y z where
    K :: x -> y -> Erk x y Char

f (K x y) = x:y
```

- \( f :: \text{Erk} a \ [a] c \rightarrow [a] \)
- \( f :: \text{Erk} a \ [\text{Char}] a \rightarrow [\text{Char}] \)
- \( f :: \text{Erk} [a] [[\text{Char}]] a \rightarrow [[[\text{Char}]]] \)
How Many Maximal Typings Can There Be?

What is the type of:

```haskell
data Erk x y z where
    K :: x \rightarrow y \rightarrow Erk x y Char
```

```haskell
f (K x y) = x:y
```

- \( f :: Erk a [a] c \rightarrow [a] \)
- \( f :: Erk a [Char] a \rightarrow [Char] \)
- \( f :: Erk [a] [[Char]] a \rightarrow [[[Char]]] \)
- \( \ldots \)
- \( f :: Erk [a]^n [Char]^{n+1} a \rightarrow [Char]^{n+1} \)
Overview

1. GADTs
   - Overview
   - Type Checking
2. The OutsideIn Algorithm
3. Formal Specification and Properties
4. Related Work
5. Summary
Inference Algorithm Requirements

- **Simple**
  - no implication constraints in types
  - *declarative* type system
- **Powerful**
  - accept as many programs with principal types as possible
- **Robust**
  - allow small program transformations
- **Efficient**
  - little/no overhead for non-GADT code
Flow of type information through the pattern match:
Flow of type information through the pattern match:

- **InsideOut**: ambiguity
- **OutsideIn**: determinism!
Our **OUTSIDEIN** algorithm:

- fix outside type information first
- before going inside
Our **OutsideIn** algorithm:

- fix **outside** type information first
- before going **inside**

Case branches become like **black holes**

- information is sucked in
- nothing escapes
Initial Program

```haskell
data Exp a where
    TRUE :: Exp Bool

size x = 1 + case x of { TRUE -> 0 }
```
OutsideIn Example

Ignore Inside

data Exp a where
  TRUE :: Exp Bool

size x = 1 + case x of { TRUE -> _ }
Infer Outside Type 1/2

data Exp a where
  TRUE :: Exp Bool

size :: Exp a -> ...
size x = 1 + case x of { TRUE -> _ }
Infer Outside Type 2/2

data Exp a where
  TRUE :: Exp Bool

size :: Exp a -> Int
size x = 1 + case x of { TRUE -> _ }
Check Inside

data Exp a where
  TRUE :: Exp Bool

size :: Exp a -> Int
size x = 1 + case x of { TRUE -> 0 }

  0 :: Int ??? YES!

just like type checking (no ambiguity)
Initial Program

data Exp a where
  TRUE :: Exp Bool

--  bad :: Exp a -> a
--  bad :: Exp a -> Bool

bad x = case x of { TRUE -> True }
Ignore Inside

data Exp a where
    TRUE :: Exp Bool

-- bad :: Exp a -> a
-- bad :: Exp a -> Bool

bad x = case x of { TRUE -> _ }
OutsideIn Ambiguous Example

Infer Outside Type 1/2

data Exp a where
  TRUE :: Exp Bool

-- bad :: Exp a -> a
-- bad :: Exp a -> Bool
bad :: Exp a -> ...
bad x = case x of { TRUE -> _ }
data Exp a where
    TRUE :: Exp Bool

-- bad :: Exp a -> a
-- bad :: Exp a -> Bool
bad :: Exp a -> b
bad x = case x of { TRUE -> _ }
Check Inside

data Exp a where
    TRUE :: Exp Bool

-- bad :: Exp a -> a
-- bad :: Exp a -> Bool
bad :: Exp a -> b
bad x = case x of { TRUE -> True }

True :: b ??? NO!
In the paper:

- formalization
- existentials, e.g., `data T a where MkT :: a -> b -> T a`
- nested let bindings
  - annotated
  - unannotated
- discriminate between ADT and GADT constructors
- minimal overhead for non-GADT code
Overview

1. GADTs
   - Overview
   - Type Checking
2. The OutsideIn Algorithm
3. Formal Specification and Properties
4. Related Work
5. Summary

Type inference for GADTs
Declarative Type Systems

Two declarative type systems:

1. the intuitive type system
   - no principal types
Two declarative type systems:

1. the intuitive type system
   - no principal types

2. our restricted type system:
   - specification of our algorithm
   - accepts the programs that our algorithm accepts
   - specification $\neq$ implementation

See draft for details.
Soundness

Unrestricted well-typing

Restricted well-typing
Incompleteness & Principality

unrestricted well-typing

restricted well-typing

principal typing
Incompleteness Example

\[(a \sim \text{Int} \Rightarrow b \sim \text{Bool})\]
Faithful implementation of restricted type system:

- sound
- complete
- terminating
- principal types
Overview

1. GADTs
   - Overview
   - Type Checking
2. The OutsideIn Algorithm
3. Formal Specification and Properties
4. Related Work
5. Summary
Related Work

1. Require full annotation
   - type checking, no ambiguity

   [Cheney&Hinze;Simonet&Pottier]
Related Work

1. Require full annotation
   - type checking, no ambiguity
   - [Cheney & Hinze; Simonet & Pottier]

2. Require entirely *un*annotated programs
   - implication constraints in signatures, checking becomes undecidable
   - [Simonet & Pottier]
Related Work

1. Require full annotation
   - type checking, no ambiguity
   [Cheney&Hinze;Simonet&Pottier]

2. Require entirely *un*annotated programs
   - implication constraints in signatures, checking becomes undecidable
   - abductive solving, incomplete algorithm
   [Simonet&Pottier]
   [Sulzmann et al.]
Related Work

1. Require full annotation [Cheney & Hinze; Simonet & Pottier]
   ▶ type checking, no ambiguity

2. Require entirely unannotated programs
   ▶ implication constraints in signatures, checking becomes undecidable [Simonet & Pottier]
   ▶ abductive solving, incomplete algorithm [Sulzmann et al.]

3. Practical compromises
   ▶ wobbly types (GHC), ad-hoc propagation [Peyton Jones et al.]
     ▶ mostly weaker than OutsideIn
     ▶ sometimes stronger: inferred type intuitively non-principal
Related Work

1. Require full annotation  
   ▶ type checking, no ambiguity  
   [Cheney & Hinze; Simonet & Pottier]

2. Require entirely *un*annotated programs  
   ▶ implication constraints in signatures, checking becomes undecidable  
   [Simonet & Pottier]  
   ▶ abductive solving, incomplete algorithm  
   [Sulzmann et al.]

3. Practical compromises  
   ▶ wobbly types (GHC), ad-hoc propagation  
     [Peyton Jones et al.]  
     ▶ mostly weaker than OutsideIn  
     ▶ sometimes stronger: inferred type intuitively non-principal  
   ▶ preprocessor framework for propagation  
     [Régis-Gianas & Pottier]  
     ▶ *Wob* instance: crf. wobbly types  
     ▶ *Ibis* instance: complex, no declarative spec., weaker?
1 GADTs
   - Overview
   - Type Checking
2 The OutsideIn Algorithm
3 Formal Specification and Properties
4 Related Work
5 Summary
You should have learned that:

- type inference for GADTs is a tricky matter
- our OutsideIn algorithm is simple yet powerful
- sound, terminating and complete wrt. its declarative type system
- derives principal solutions in its own type system and the generally ambiguous type system
Type Class Constraints

```haskell
class Eq a where { (==) :: a -> a -> Bool }

data T a = Eq a => MkT

f x y = not (case x of { MkT -> y==y })
```

What is the signature of `f`?
Type Class Constraints

class Eq a where { (==) :: a -> a -> Bool }

data T a = Eq a => MkT

f x y = not (case x of { MkT -> y==y })

What is the signature of f?

\[ f :: \text{Eq } b \Rightarrow \text{T } a \rightarrow b \rightarrow \text{Bool} \]
Type Class Constraints

```haskell
class Eq a where { (==) :: a -> a -> Bool }

data T a = Eq a => MkT

f x y = not (case x of { MkT -> y==y })
```

What is the signature of \( f \)?

- \( f :: Eq b => T a -> b -> Bool \)
- \( f :: T a -> a -> Bool \)
Other Constraints, Same Issues

Lack of principal typing arises for:

- equality constraints
- type class constraints
- equality constraints involving type families
- implicit parameters

Our algorithm works for these cases too.
Stratification of Constraints

Type class constraints AND equality constraints:

- **Given** equality constraints **do help** solve **wanted** type class constraints.
Type class constraints AND equality constraints:

- **Given** equality constraints do help solve **wanted** type class constraints.

  \[ a \sim \text{Int} \Rightarrow \text{Eq } a \]
Type class constraints AND equality constraints:

- **Given** equality constraints do help solve **wanted** type class constraints.
  
  \[ a \sim \text{Int} \Rightarrow \text{Eq} \ a \]  
  
  \[ \rightarrow \text{use OutsideIn strategy} \]
Type class constraints AND equality constraints:

- **Given** equality constraints do help solve **wanted** type class constraints.

\[ a \sim \text{Int} \Rightarrow \text{Eq} \ a \]

\[ \rightarrow \text{use OutsideIn strategy} \]

- **Given** type class constraints do **not help** solve **wanted** equality constraints.
Type class constraints AND equality constraints:

- **Given** equality constraints do help solve **wanted** type class constraints.
  
  \[ a \sim \text{Int} \Rightarrow \text{Eq } a \]

  \[ \rightarrow \text{use OutsideIn strategy} \]

- **Given** type class constraints do not help solve **wanted** equality constraints.

  \[ \text{Eq } a \Rightarrow a \sim \text{Int} \]
Type class constraints AND equality constraints:

- **Given** equality constraints do help solve **wanted** type class constraints.
  
  \[
  a \sim \text{Int} \Rightarrow \text{Eq } a
  \]

  \[\rightarrow\] use **OutsideIn** strategy

- **Given** type class constraints do not help solve **wanted** equality constraints.
  
  \[
  \text{Eq } a \Rightarrow a \sim \text{Int}
  \]

  \[\rightarrow\] float wanted equality constraint out
Type class constraints AND equality constraints:

- **Given** equality constraints do help solve **wanted** type class constraints.
  
  \[ a \sim \text{Int} \rightarrow \text{Eq } a \]

  \[ \rightarrow \text{use } \text{OutsideIn} \text{ strategy} \]

- **Given** type class constraints do not help solve **wanted** equality constraints.
  
  \[ \text{Eq } a \rightarrow a \sim \text{Int} \]

  \[ \rightarrow \text{float wanted equality constraint out} \]

[Work in progress.]
Questions?

Further reading: