

Type system support for GADTs and some ideas towards practical dependently typed programming

Dimitrios Vytiniotis

Computer and Information Science Department
University of Pennsylvania

Presentation at National Technical University of Athens, December 2006

Material partially based on:

“Simple, unification-based type inference for GADTs” (*ICFP'06*)

with Simon Peyton Jones, Stephanie Weirich, and Geoff Washburn

Functional programming with internal verification

- First type systems (e.g. the ML type system) reason for existence:
 - ▶ Ensure crash-free program behaviour without runtime tests.
- Modern view: type systems for verification of **correctness properties**.
 - ▶ Via Curry-Howard correspondence types express properties.
 - ▶ Specifications written in the types (alas internal).
 - ▶ Proofs are programs, properties statically verified via type checking.

Expressive type systems:

A form of **lightweight internal verification**; a process meant to co-exist with external verification.

The Haskell view on dependently typed programming

Pseudo-dependently typed programming used **all the time!**

- Phantom types, higher-rank types, type classes with functional dependencies, **generalized algebraic data types**.

The Haskell view on dependently typed programming

Pseudo-dependently typed programming used **all the time!**

- Phantom types, higher-rank types, type classes with functional dependencies, **generalized algebraic data types**.
- Complete type inference for such extensions is often undecidable.
- Goal: must not sacrifice type inference for the whole language.
- Solution: Take advantage of user type annotations, in a mixture of type checking and type inference.

The Haskell view on dependently typed programming

Pseudo-dependently typed programming used **all the time!**

- Phantom types, higher-rank types, type classes with functional dependencies, **generalized algebraic data types**.
- Complete type inference for such extensions is often undecidable.
- Goal: must not sacrifice type inference for the whole language.
- Solution: Take advantage of user type annotations, in a mixture of type checking and type inference.

This presentation: Incorporating `GADTs` in the type inference engine of a **practical** programming language. Implemented in the current version of the Glasgow Haskell Compiler.

Generalized algebraic data types

The unit conversion problem, in Haskell'98:

```
type Meters = Double
type Feet   = Double

add :: Double -> Double -> Double
add x y = x + y  -- terrible!
```

Or, better:

```
newtype Meters = M Double
newtype Feet   = F Double

addMeter :: Meters -> Meters -> Meters
addMeter (M x) (M y) = M (x+y)

addFeet :: ... -- similarly!
```

Generalized algebraic data types

Or, using a single datatype:

```
data Unit = M Double
          | F Double

add (M x) (M y) = M (x+y)
add (F x) (F y) = F (x+y)
add _ _ = error "can't convert units!"
```

But the consumer of this code may attempt to call the function with wrong arguments. The GADT solution:

```
data Meters; data Feet;
data Unit a where
  M :: Double -> Unit Meters
  F :: Double -> Unit Feet

add :: forall a. Unit a -> Unit a -> Unit a
add (M x) (M y) = M (x+y)
add (F x) (F y) = F (x+y) -- no need for the rest, unreachable branches!
```

Compare the two definitions of Unit!

The most famous GADT: typed abstract syntax

Consider a conventional interpreter:

```
data Term = BLit Bool | ILit Int | If Term Term Term | Suc Term

eval :: Term -> Term
eval (If t1 t2 t3) =
  case (eval t1) of
    BLit True  -> eval t2
    BLit False -> eval t3
    _         -> error "must not happen!"
eval (Suc t) =
  case eval t of
    ILit i -> ILit (i+1)
    _     -> error "must not happen!"
```

Several problems:

- Can't detect and prevent the pattern match failures.
- A lot of inefficient tagging and pattern matching.

The most famous GADT: typed abstract syntax

```
data Term a where
  BLit :: Bool -> Term Bool
  ILit :: Int -> Term Int
  If    :: Term Bool -> Term a -> Term a -> Term a
  Suc   :: Term Int -> Term Int

eval :: forall a. Term a -> a
eval (BLit b) = b           -- a = Bool
eval (ILit i) = i           -- a = Int
eval (If t1 t2 t3) = if (eval t1) then (eval t2) else (eval t3)
eval (Suc t) = 1 + eval t   -- a = Int
```

- The variable `a` in `Term a` is not a parameter, but rather an `index`.
- Return types of constructors not uniform.
- Pattern matching introduces refinement.

Type inference for GADTs is problematic

Lack of principal type to infer for a function and use throughout its scope:

- Threatens completeness!

Consider:

```
eval (ILit i) = i
eval (Suc t) = 1 + eval t
```

What type we infer for `eval`?

- $\forall a. \text{Term } a \rightarrow \text{Int}$
- $\text{Term Int} \rightarrow \text{Int}$
- $\forall a. \text{Term } a \rightarrow a$

Our solution:

- Allow refinement to happen only when types originate in some user type annotation; otherwise treat patterns as ML.
- Implies that the program above gets **only** $\text{Term Int} \rightarrow \text{Int}$.

Wobbly and rigid types

We need to separate types:

- Rigid types; originating in some user annotation.
 - ▶ If one thinks operationally, they cannot contain unification variables.
- Wobbly types; containing guessed information.
 - ▶ Operationally, may contain unification variables.

Two judgements:

$$\Gamma \vdash e :^w \tau \quad \text{and} \quad \Gamma \vdash e :^r \tau$$

The rules for these judgements maintain the wobbly-rigid invariant:

$$\frac{\Gamma \vdash e_1 :^w \tau_1 \quad \Gamma \vdash e_2 :^w \tau_1 \rightarrow \tau_2}{\Gamma \vdash e_1 e_2 :^m \tau_2} \qquad \frac{\Gamma, x :^m \tau_1 \vdash e :^m \tau_2}{\Gamma \vdash \lambda x. e :^m \tau_1 \rightarrow \tau_2}$$

Rigidity can be traced down to annotations

Rigidity is originating in some user annotation:

$$\frac{\Gamma \vdash u :^r \sigma \quad \Gamma, x :^r \sigma \vdash e :^m \tau}{\Gamma \vdash \text{let } x :: \sigma = u \text{ in } e :^m \tau} \qquad \frac{\Gamma \vdash u :^r \tau}{\Gamma \vdash u :: \tau :^m \tau}$$

Moreover, we have a way of inferring the **rigidity** of a given type:

$$\frac{(x :^m \tau) \in \Gamma}{\Gamma \vdash x : \tau \succ m} \qquad \frac{\Gamma \vdash e :^m \tau}{\Gamma \vdash e : \tau \succ w} \qquad \frac{\Gamma \vdash e :^r \tau}{\Gamma \vdash (e :: \tau) : \tau \succ r}$$

Rigidity can be traced down to annotations

Rigidity is originating in some user annotation:

$$\frac{\Gamma \vdash u :^r \sigma \quad \Gamma, x :^r \sigma \vdash e :^m \tau}{\Gamma \vdash \text{let } x :: \sigma = u \text{ in } e :^m \tau} \qquad \frac{\Gamma \vdash u :^r \tau}{\Gamma \vdash u :: \tau :^m \tau}$$

Moreover, we have a way of inferring the **rigidity** of a given type:

$$\frac{(x :^m \tau) \in \Gamma}{\Gamma \vdash x : \tau \succ m} \qquad \frac{\Gamma \vdash e :^m \tau}{\Gamma \vdash e : \tau \succ w} \qquad \frac{\Gamma \vdash e :^r \tau}{\Gamma \vdash (e :: \tau) : \tau \succ r}$$

... and this last procedure can be made more or less precise, this is **not** the important part!

The important part: simple, unification-based refinement!

All magic lies in pattern matching expressions:

$$\begin{array}{lcl} \text{case } e \text{ of} & J_1 \bar{x} & \rightarrow e_1 \\ & \dots & \rightarrow \dots \\ & J_n \bar{y} & \rightarrow e_n \end{array}$$

Call e , the **scrutinee** of the *case* expression. Two basic principles:

- 1 If the scrutinee type is unannotated do not use refinement; treat pattern matching **as in ML!**.
- 2 Subtle: Apply refinement only to rigid parts of judgement.

Principle 1: Do not use refinement if scrutinee is wobbly!

It is threatening type inference completeness:

```
data T a where TInt  :: Int -> T Int
                  TBool :: Bool -> T Bool

foo = let bar = fun x ->
          case x of TInt y -> y
                in (bar (TBool True) , bar (TInt 42))
```

It's the same example from the introduction, really:

- Type system gives *bar* the type $\forall a. T a \rightarrow a$, and succeeds.
- Any simple algorithm would go with the more specialized type $T Int \rightarrow Int$. The algorithm would fail!

Completeness is gone!

Principle 2: Never apply refinement to wobbly types!

It is threatening type inference completeness (same old problem):

```
data T a where MkT :: T Int

foo :: forall a. T a -> Int
foo x = let _ = fun z ->
          (fun _ -> (case x of MkT -> [x,z])) [x,z]
      in 42
```

- First, $x: ^r T a$, $z: ^w \tau$ in context.
- Next, refinement of a to Int happens, since scrutinee is rigid.
 - ▶ Type system: could guess that a for τ , and check both $[x,z]$'s, inner one typed as $[Int]$, outer one typed as $[a]$.
 - ▶ **But**, the algorithm could only put a unification variable there for τ , then apply the refinement for x and infer that $\tau = Int$. This would immediately make the outer $[x,z]$ to fail!

Conclusion: We must not apply refinement if the return type of the case is wobbly (as is here!), in order to achieve a complete type inference search-free, unbiased algorithm!

Type checking case expressions with wobbly scrutinee

$$\frac{\Gamma \vdash e : \tau_s \succ m_s \quad \Gamma \vdash J_i \bar{x} \rightarrow e_i :^{m_s, m} \tau_s \rightarrow \tau \text{ for all } i}{\Gamma \vdash \text{case } e \text{ of } \{J \bar{x} \rightarrow e\} :^m \tau}$$

If the scrutinee is wobbly, no refinement is happening—just as in Haskell:

$$\frac{\begin{array}{l} J :^r \forall \bar{a}. \bar{\tau}_1 \rightarrow T \bar{\tau}_2 \in \Gamma \quad \bar{a} \# \text{ftv}(\Gamma, \bar{\tau}_p, \tau_t) \\ \bar{a}_c = \bar{a} \cap \text{ftv}(\bar{\tau}_2) \quad \theta = [\bar{a}_c \mapsto v] \quad \theta(\bar{\tau}_2) = \bar{\tau}_p \\ \Gamma, x :^w \theta(\bar{\tau}_1) \vdash t :^m \tau_t \end{array}}{\Gamma \vdash J \bar{x} \rightarrow e :^{w, m} T \bar{\tau}_p \rightarrow \tau_t}$$

Type checking case expressions with rigid scrutinees

If the scrutinee is rigid, use refinement, respecting the previous principles:

$$\frac{J :^r \forall \bar{a}. \overline{\tau_1} \rightarrow T \overline{\tau_2} \in \Gamma \quad \bar{a} \# ftv(\Gamma, \overline{\tau_p}, \tau_t) \quad \theta \in \text{fmgu}(\overline{\tau_p} \doteq \tau_2) \quad \theta(\Gamma, \overline{x :^r \tau_1}) \vdash t :^m \theta^m(\tau_t)}{\Gamma \vdash J \overline{x} \rightarrow e :^{r,m} T \overline{\tau_p} \rightarrow \tau_t}$$

- Refinement expressed as a most general unifier.
- Refinement is **selectively applied**. Notice

$$\theta(\Gamma, \overline{x :^r \tau_1}) \quad \text{and} \quad \theta^m(\tau_t)$$

Type checking case expressions with rigid scrutinee

If the scrutinee is rigid, use refinement, respecting the previous principles:

$$\frac{J :^r \forall \bar{a}. \overline{\tau_1} \rightarrow T \overline{\tau_2} \in \Gamma \quad \bar{a} \# ftv(\Gamma, \overline{\tau_p}, \tau_t) \quad \theta \in \text{fmg}u(\overline{\tau_p} \doteq \tau_2) \quad \theta(\Gamma, \overline{x :^r \tau_1}) \vdash t :^m \theta^m(\tau_t)}{\Gamma \vdash J \overline{x} \rightarrow e :^{r,m} T \overline{\tau_p} \rightarrow \tau_t}$$

- Refinement expressed as a most general unifier.
- Refinement is **selectively applied**. Notice

$$\theta(\Gamma, \overline{x :^r \tau_1}) \quad \text{and} \quad \theta^m(\tau_t)$$

Remarks:

- It is sound to use a most general unifier, or any pre-unifier!
- Is it sound to use any unifier? (No)
- What if there is no unifier?

Still trouble: How many *mgus* are there?

```
data Eq a b where Refl :: forall c. Eq c c

f :: forall a b. Eq a b -> (a -> Int) -> b -> Int
f x y z = (fun w -> case x of Refl -> y w) z
```

- x, y, z with rigid types in environment, refinement in pattern matching, then applied to rigid environment:

$$\theta \in \text{mgu}(Eq\ c\ c \doteq Eq\ a\ b)$$

But there are three of those unifiers:

- ▶ $\theta = [a \mapsto c, b \mapsto c]$
- ▶ $\theta = [a \mapsto b, c \mapsto b]$
- ▶ $\theta = [b \mapsto a, c \mapsto a]$

But the type of z outside the pattern match is simply b and therefore **only the second unifier will succeed!**

Terrible news! We lose completeness of the algorithm again.

Solution: fresh mgus

A **fresh mgu** or *fmgu* is a unifier where such ambiguities are resolved with the introduction of fresh variables, and no spurious equalities are justified. Almost like an *mgu* but with more completely fresh variables.

- Intuitively, these fresh variables express a uniquely chosen representative of each equivalence class.
- E.g only $\theta = [a \mapsto c, b \mapsto c]$ is an *fmgu* of $\text{Eq } c \doteq \text{Eq } a \ b$.
- *fmgu*s solve the directionality problem, and are still sound! (why?)

More technical details in the paper or the technical report.

Summary of contributions of this work

- Proofs of elaboration-based type soundness.
- Existence of principal types and a complete type inference algorithm.
- Treatment of lexically-scoped type variables in annotations.
- Treatment of nested patterns.

The paper is now implemented in GHC! Pattern matching with unification is a very delicate technical subject! Our counterexamples and design choices were driven by horrible proof failures!

Most closely related work is

“Stratified type inference for generalized algebraic datatypes” (*POPL'06*)

by François Pottier and Yann Régis-Giannas.

Dependently typed programming with GADTs

GADTs for dependently typed programming or theorem proving:

```
data S a; data Z;  
data DList n where  
  Nil :: DList Z  
  Cons :: forall m. Double -> List m -> List (S m)  
  
safehead :: forall m. List (S m) -> Double  
safehead (Cons d l) = d
```

What is missing?

But consider the *append* function:

```
data Plus n m q where
  Base :: forall m. Plus Z m m
  Ind  :: Plus n m q -> Plus (S n) m (S q)

append :: Plus n m q -> List n -> List m -> List q
append Base Nil l = l
append (Ind pr) (Cons d l0) l = Cons d (append pr l0 l)
```

Works, but a little unsatisfactory: The user did all the proofs! Haskell hackers today try to eliminate the *Plus n m q* predicate with type classes with functional dependencies:

```
append :: Plus n m q => List n -> List m -> List q
```

But the interaction of GADTs and type classes is not well-specified (work on this is ongoing)

True type functions

Perhaps what is missing, is **true type functions!**

```
typedefun Add :: * -> * where
Add Z m = m
Add (S n) m = S (Add n m)

append :: List n -> List m -> List (Add n m)
append Nil l = l
append (Cons d l0) l = Cons d (append l0 l)
```

But there are problems:

- Higher-order unification!
- How does HOU interact with GADT refinement?
- Type functions must define a confluent (for type inference completeness) and normalizing (for decidability) rewrite system.

First-class existential types

Another common style of dependently typed programming is returning values together with proofs about their properties:

$$\text{append} :: \forall nm. \text{List } n \rightarrow \text{List } m \rightarrow \exists p. (\text{List } p, \text{Plus } n \ m \ p)$$

Good, only GHC does not support first class existentials!

```
data FakeEx n m where
  MkF :: forall p n m. (List p, Plus n m p) -> FakeEx n m

append :: List n -> List m -> FakeEx n m
append Nil l = MkF (l, Base)
append (Cons d l0) l = case (append l0 l) of
  MkF (al,prf) -> MkF (Cons d al, Ind prf)
```

Type inference figures out all types for us, but the fact that we used a specially introduced existential type for this instead of built-in is terribly annoying!

Work plan

- Type-level computations.
 - ▶ Start point: a dependently typed language.
 - ▶ Type system tracking termination behaviour of type-indexing functions.
 - ▶ Exploring pattern matching that introduces only unifiers with first-order domains. Open question: Can we still make any use of the second-order constraints in the right-hand sides?
- First-class existentials.
 - ▶ Extension of first-class polymorphism support (Boxy Types, MLF) to existential types.
 - ▶ Very tricky, with a lot of design choices that must be evaluated.

Related

- Coq may actually be quite suitable for development!
 - ▶ But the lack of context refinement is often unsatisfactory.
 - ▶ World effects?
- Tim Sheard's Ω mega is a nice attempt in addressing the first point above. Takes the Haskell approach of type-level computations; instead of the dependent types avenue.
- Epigram? Agda? *RSP1*?

Thank you for your attention!